# A Large Empirical Assessment of the Role of Data Balancing in Machine-Learning-based Code Smell Detection

Fabiano Pecorelli[a], Dario Di Nucci[b], Coen De Roover[c], Andrea De Lucia[a]

[a] *SeSa Lab - University of Salerno, Fisciano, Italy*
[b] *Tilburg University - Jheronimus Academy of Data Science, 's-Hertogenbosch, The Netherlands*
[c] *Vrije Universiteit Brussel, Brussels, Belgium*

**Abstract**

Code smells can compromise software quality in the long term by inducing technical debt. For this reason, many approaches aimed at identifying these design flaws have been proposed in the last decade. Most of them are based on heuristics in which a set of metrics is used to detect smelly code components. However, these techniques suffer from subjective interpretations, a low agreement between detectors, and threshold dependability. To overcome these limitations, previous work applied Machine-Learning that can learn from previous datasets without needing any threshold definition. However, more recent work has shown that Machine-Learning is not always suitable for code smell detection due to the highly imbalanced nature of the problem. In this study, we investigate five approaches to mitigate data imbalance issues to understand their impact on Machine Learning-based approaches for code smell detection in Object-Oriented systems and those implementing the Model-View-Controller pattern. Our findings show that avoiding balancing does not dramatically impact accuracy. Existing data balancing techniques are inadequate for code smell detection leading to poor accuracy for Machine-Learning-based approaches. Therefore, new metrics to exploit different software characteristics and new techniques to effectively combine them are needed.

*Keywords:* Code Smells, Machine Learning, Data Balancing, Object Oriented, Model View Controller

## 1. Introduction

During software development, strict deadlines and new requirements could lead to the introduction of *technical debt* [1], namely a set of design issues that may negatively affect the system's maintainability in the future. *Code smells* [2] are one of the first indications of code technical debt, i.e., sub-optimal design solutions that developers apply to a software system.

Code smells have been investigated from several perspectives [3, 4]: their introduction [5, 6] and evolution [7–9], their impact on reliability [10, 11] and maintainability [12, 13], as well as the way developers perceive them [14–16] have been deeply analysed in literature and have revealed that code smells represent serious threats to source code maintenance and evolution.

For all these reasons, several techniques to automatically identify code smells in source code have been investigated [17–19]. These techniques rely on heuristics and discriminate code artefacts affected (or not) by a specific type of smell through the application of detection rules that compare the values of relevant metrics extracted from source code against empirically identified thresholds. The accuracy of such approaches has been empirically assessed and was found to be reasonably high. Nevertheless, they share limitations that hinder their adoption in practice [17, 20]. First, they might return code smell candidates that are not considered as actual problems by developers [21, 22]. Furthermore, the agreement between detectors is very low [23], which means that different detectors are required to detect the smelliness of various code components. Finally, the accuracy of most of the current detectors is strongly influenced by the thresholds needed to identify instances of the smells. [17].

To overcome these limitations, researchers recently adopted Machine-Learning (ML) to avoid thresholds and decrease the false positive rate [24]. In this approach, a classifier is trained on previous releases of the source code by exploiting a set of independent variables (e.g., structural, historical, or textual metrics). The resulting model is employed to determine the presence of smells or the degree of smelliness of a code element. Although the use of Machine-Learning looks promising, previous work has observed contradicting results [24–26]. Heuristics-based approaches perform slightly better than machine learning approaches, thus indicating that Machine-Learning is still unsuitable for code smell detection [26]. As code smell detection is a problem in which training datasets usually have skewed class proportions (i.e., highly data imbalanced) [25, 26], data balancing is a key factor to improve the reliability of such models. Data balancing can be introduced in several ways by transforming the training set or by using cost-sensitive classifiers.

In this paper, we extend our previous work [27] whose

*Published in: Journal of Systems and Software.*
*This version of the article has been accepted for publication, after peer review and is subject to Elsevier terms of use.*
*The Version of Record is available online at: https://dx.doi.org/10.1016/j.jss.2020.110693*

results suggested several advantages and disadvantages in applying data balancing techniques that eventually do not dramatically improve the accuracy of the models. We make a step further by proposing a more extensive empirical study in which we compare the performance of five data-balancing techniques for code smell detection with respect to a *no-balancing* baseline. To increase the generalisability of the results, we analyse two subsets of code smells extracted from two catalogues: (i) the catalogue proposed by Fowler et al. [2] for Object-Oriented code, and (ii) the catalogue proposed by Aniche et al. [28] for systems implementing the Model-View-Controller pattern. Our goal is understanding to what extent data balancing techniques can improve the accuracy of Machine-Learning for code smell detection and which algorithms practitioners should use. This paper extends our previous conference publication [27] by adding the following contributions:

1. We expand the study on code smells for Object-Oriented systems by considering six additional code smells (i.e., Feature Envy, Inappropriate Intimacy, Middle Man, Refused Bequest, Speculative Generality, Long Parameter List). Thus, overall, we use Machine-Learning-based techniques to detect 11 code smell types on 125 releases of 13 software systems.

2. We report a new empirical study that includes four code smells to detect maintainability issues in Model-View-Controller systems [28]. Specifically, we analyse 120 projects relying on the Spring framework to answer two additional research questions.

3. We further analyse the role of balancing techniques and the impact of metrics selection.

4. We inspect the overhead in terms of efficiency caused by data balancing.

5. We provide a comprehensive replication package containing the raw data and scripts used to carry out the empirical study [29].

The results suggest that balancing does not sensibly improve performance. Techniques which perform training only on the minority class (i.e., *Cost-Sensitive Classifier* and *One-Class Classifier*), and resampling techniques (i.e., *Oversampling* and *Undersampling*) are both not effective. Creating synthetic instances (i.e., *SMOTE*) is effective but not applicable in some cases due to the low number of smelly instances. Therefore, existing data balancing techniques are inadequate for code smell detection. This consideration hinders the feasibility of the current Machine-Learning-based approaches and shows that further work is needed to achieve automated code smell detection. In particular, new metrics [30, 31] and techniques able to effectively combine them with structural metrics are needed.

*Structure of the paper.* In Section 2, we discuss the literature related to Machine-Learning-based code smell detection, and data balancing techniques. In Section 3, we replicate the empirical study presented in [27] on an extended set of code smells proposed by Fowler [2] for Object-Oriented systems.

In Section 4, we present the new empirical study on code smells specific for systems implementing the Model-View-Control pattern [28]. Section 5 discusses the results of the two, while in Section 6 we sketch possible threats to validity. Finally, Section 7 concludes the paper.

## 2. Related Work

In this section, we describe the related work concerning *Machine-Learning for code smell detection* and *the impact of data balancing techniques*.

### *2.1. Machine-Learning for Code Smell Detection*

Machine-Learning (ML) has been used in several recent works on code smell detection [4]. Kreimer [34] proposed a prediction model based on *Decision Trees* and code metrics to detect two code smells (i.e., *Blob* and *Long Method*). This model can lead to high values of accuracy. Later on, Amorim et al. [35] confirmed the previous findings on four medium-scale open-source projects. Vaucher et al. [36] studied *Blob*'s evolution relying on a *Naive Bayes* classifier, whereas Maiga et al. [37] proposed the use of *Support Vector Machine* (SVM). The use of *Bayesian Belief Networks* to detect *Blob*, *Functional Decomposition*, and *Spaghetti Code* instances on open-source programs, proposed by Khomh et al. [38] lead to an overall F-Measure close to 60%. Similarly, Hassaine et al. [39] defined an immune-inspired approach for the detection of *Blob* smells, while Oliveto et al. [40] used B-Splines to detect them. Arcelli Fontana et al. made the most relevant progress in this field [24, 41, 42]. In their work, they (i) theorised that ML might lead to a more objective evaluation of the smells' hazardousness [41], (ii) provided a ML method to assess code smell intensity [42], and (iii) compared 16 ML techniques for the detection of four code smell types [24] showing that ML can lead to F-Measure values close to 100%. Nevertheless, recently Di Nucci et al. [25] demonstrated that, in a real use-case scenario, the results achieved by Arcelli Fontana et al. [24] could not be generalised, thus casting doubt on the actual effectiveness of machine learning for code smell detection. Finally, Pecorelli et al. [26] compared ML-based and heuristic metric-based approaches to assess the real capabilities of ML in the context of code smell detection showing that heuristic techniques for code smell detection still perform slightly better.

### *2.2. The Impact of Data Balancing Techniques*

Imbalanced learning concerns learning from datasets where some classes are underrepresented. Despite many real-world Machine-Learning applications, learning from

| Code Smell | Short Description | Detection Rule | ML Model Features |
|---|---|---|---|
| God Class | This smell characterises classes having a large size, poor cohesion, and several dependencies on other data classes of the system [2] | $ELOC > \alpha \wedge (WMC + NOA) > \beta \wedge LCOM > \gamma$ | $ELOC$, $WMC$, $NOA$, $LCOM$ |
| Spaghetti Code | Classes affected by this smell declare several long methods without parameters [2] | $ELOC > \alpha \wedge NMNOPARAM > \beta$ | $ELOC$, $NMNOPARAM$ |
| Class Data Should Be Private | This smell appears in cases where a class exposes its attributes, thus violating the information hiding principle [2] | $NOPA > \alpha$ | $NOPA$ |
| Complex Class | Classes presenting an overly high cyclomatic complexity [32] are affected by this design flaw | $McCabe > \alpha$ | $McCabe$ |
| Long Method | Methods implementing more than one functionality are affected by this smell [2] | $LOC\_METHOD > \alpha \wedge NP \geq \beta$ | $LOC\_METHOD$, $NP$ |
| Feature Envy | This smell arises when a method communicates more with methods that are inside another class than the ones in its class [2] | $MC > \alpha \wedge ATFD > \beta$ | $MC$, $ATFD$ |
| Inappropriate Intimacy | This smell occurs when two classes are highly coupled [18, 33] | $(FanIn + FanOut) > \alpha$ | $FanIn$, $FanOut$ |
| Middle Man | This smell arises when a class delegates to other classes most of the methods it implements [2] | $PDM > \alpha$ | $PDM$ |
| Refused Bequest | A class which redefines most of its inherited methods, then making the hierarchy wrong [2] | $PRM > \alpha$ | $PRM$ |
| Speculative Generality | This smell shows up when a class declared as abstract has very few children using its methods [2] | $NOC > \alpha$ | $NOC$ |
| Long Parameter List | A method having a long list of parameters is harder to use [2] | $NP > \alpha$ | $NP$ |

Table 1: Descriptions of Object-Oriented code smells along with the heuristics used to detect them and the features used by the ML models

| Acronym | Full Name | Smells |
|---|---|---|
| ATFD | Access To Foreign Data | Feature Envy |
| ELOC | Effective Lines Of Code | God Class, Spaghetti Code |
| FanIn | Max number of references to the subject class from another class in the system | Inappropriate Intimacy |
| FanOut | Max number of references from the subject class to another class in the system | Inappropriate Intimacy |
| LCOM | Lack of COhesion in Methods | God Class |
| LOC_METHOD | Lines Of Code of METHOD | Long Method |
| McCabe | McCabe's Cyclomatic Complexity | Complex Class |
| MC | Method Calls | Feature Envy |
| NOA | Number Of Attributes | God Class |
| NOC | Number Of Children | Speculative Generality |
| NOM | Number Of Methods | God Class |
| NOPA | Number Of Public Attributes | Class Data Should Be Private |
| NP | Number of Parameters | Long Method, Long Parameter List |
| NMNOPARAM | Number of Methods with NO PARAMeters | Spaghetti Code |
| PDM | Percentage of Delegated Methods | Middle Man |
| PRM | Percentage of Refused Methods | Refused Bequest |
| WMC | Weighted Methods Count | God Class, Complex Class |

Table 2: Complete list of the considered metrics for the detection of Object-Oriented code smells.

imbalanced data is still not trivial. Unfortunately, in many applications, these minority classes are usually also the ones of interest [25, 43–45]. Batista et al. [46] provide evidence that data sampling can be used to avoid the side-effects of data imbalance. In particular, oversampling methods are more effective than under-sampling methods in terms of prediction accuracy. Chawla [47] and He et al. [48] discussed and compared several sampling techniques used for data balancing, whereas Dittman et al. [49, 50] exploited the combination of future selection and data sampling on bioinformatics datasets. The latter found random undersampling is more computationally efficient than other sampling algorithms, including "no-sampling", although not more effective. Generally, two of the most compelling questions when dealing with data imbalance are class distribution [51] and data sparsity within each class [52]. Indeed, although this assumption does not hold for some problems [53], many classifiers assume normality in the data distribution. To solve the aforementioned issues, one-class learners, wherein the classifier learns on the target class alone, are an interesting alternative to traditional discriminative approaches [54]. Another possible solution is combining data sampling and ensemble techniques. Galar et al. [55] analysed the performance of ensemble classifiers on imbalanced datasets. Their comparison has shown that approaches combining undersampling techniques with boosting [56] or bagging [57] perform better, therefore justifying the increasing complexity through significant enhancements. A similar study was conducted by Khoshgoftaar et al. [58] who experimented boosting with three weak learners on six high-dimensional imbalanced bioinformatics datasets. Their results report that the combination of data sampling and boosting technique can lead to statistically significant results with re-

spect to only data sampling. Finally, Ditman et al. [59, 60] experimented the combination of data sampling and Random Forest [61]. Their results show that this ensemble technique is fairly robust on imbalanced data and adding data balancing does not positively contribute to the performance in a statistically significant manner.

## 3. Detection of Object-Oriented Code Smells

The purpose of this study is to understand the impact of data balancing techniques on the accuracy of Machine-Learning algorithms in detecting the design flaws from the catalogue designed by Fowler [2] who introduced the term code smell and adopted it for Object-Oriented systems. We aim to address the following research questions:

**RQ1.** Do data balancing techniques improve the effectiveness of Machine-Learning-based detectors of code smell defined for Object-Oriented systems?

**RQ2.** Which data balancing technique is the most effective at improving the effectiveness of Machine-Learning-based code smell detectors for Object-Oriented systems?

### *3.1. Code Smells for Object-Oriented systems*

Code smells are "symptoms of poor design and implementation choices" [2] that have been widely observed to both analyse their characteristics [5, 6, 62–65] and assess their impact on software maintainability [14, 16, 66–70]. For many of these code smells heuristic detection rules have been defined [2, 18, 32, 33] based on metrics and thresholds to discriminate whether a component is smelly or not. We use the same metrics used by these heuristic detection rules to build machine learning models for code smell detection. In particular, we consider 11 code smells defined by Fowler [2] that are reported in Table 1 along with their descriptions, detection rules, and lists of the metrics used in the Machine Learning models. The full description of such metrics is shown in Table 2. With respect to our previous submission [27], we analyse six new code smells (i.e., Feature Envy, Inappropriate Intimacy, Middle Man, Refused Bequest, Speculative Generality, Long Parameter List).

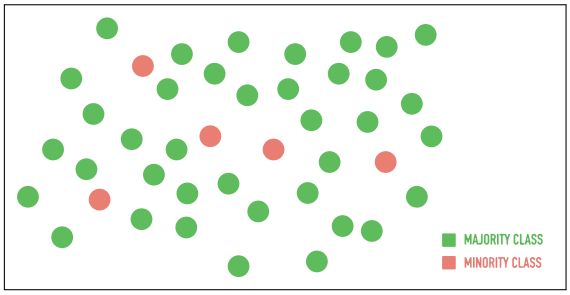


Figure 1: An example of unbalanced dataset

### *3.2. Data Balancing Techniques for Machine Learning*

The goal of the experiment is to compare the accuracy of different data balancing techniques. To this aim, we configure five different model variants based on the Naive Bayes classifier [71] which in our previous study [26] showed to be the most effective in code smell detection. Our baseline consists of models trained without applying any data balancing technique (*No-balancing*). A dataset is imbalanced when its classes are not equally represented. Figure 1 plots a simplified representation of an imbalanced dataset in which most of the instances belong to the *green* majority class.

Data balancing can be introduced by resampling/transforming the training set or by using meta-classifiers (e.g., cost-sensitive classifiers):

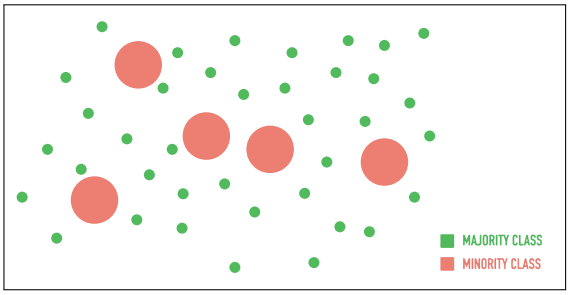


Figure 2: Example of application of Oversampling

*Oversampling [72].* This algorithm randomly adds samples of the minority class. Figure 2 shows a representation of the effects of the algorithm. In our experiment, we rely on ClassBalancer, an oversampling implementation provided in Weka [73]: the instances in the training set are re-weighted so that the sum of the weights for each class is equal. In other words, instances are not added or removed, but their weights are modified in such a way that more importance is given to the instances belonging to the minority class.

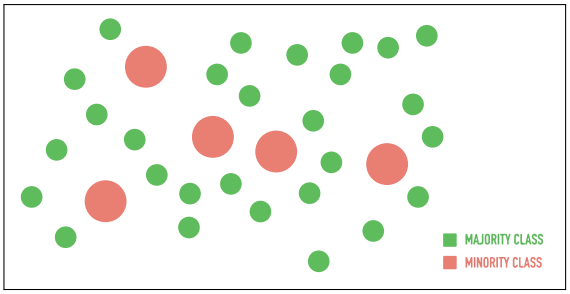


Figure 3: Example of application of Undersampling

*Undersampling.* This algorithm randomly removes samples of the majority class using either sampling with or without replacement. In our experiment, we replace instances of the majority class (i.e., clean classes) with instances from the minority class (i.e., smelly classes) until

obtaining an even number of instances for both classes as shown in Figure 3. Please notice that in the figure, the size of a point represents its frequency. In other words, as suggested by several studies from the state-of-the-art [50, 55], we undersampled the majority class with replacement. We rely on ReSample, an implementation provided in Weka [73].

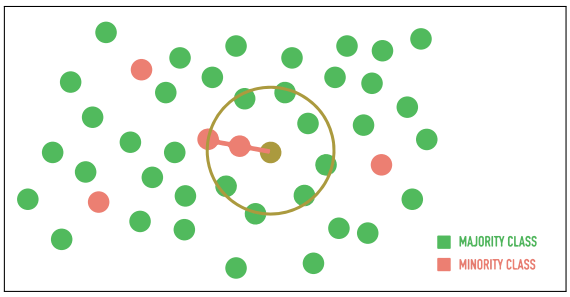


Figure 4: Example of application of SMOTE

*Synthetic Minority Oversampling TEchnique [74].* This technique increases the number of instances from the minority class by generating new synthetic instances based on the nearest neighbours belonging to that class. As shown in Figure 4, to create a new synthetic instance, *SMOTE* randomly selects an element from the minority class and identifies its nearest neighbours: the new instance is created between them. The number of nearest neighbours to use is a parameter of the algorithm. Lack of such instances causes the algorithm to fail, as explained in Section 5. To experiment with this technique, we rely on the implementation provided by Weka [73]. To reduce the algorithm failures, we set the number of neighbours to the minimum allowed value (i.e., two).

*Cost-Sensitive Classifier [75].* A Cost-Sensitive Classifier is a meta-classifier that renders a cost-sensitive version of the base classifier. The training instances can be reweighted according to the total cost assigned to each class, i.e., the cost-sensitivity is considered during the training phase. Considering that ML-based code smell detection exhibits many false negatives, we configure the CostSensitiveClassifier provided by Weka [73] in such a way that the cost of false negatives is twice the cost of false positives.

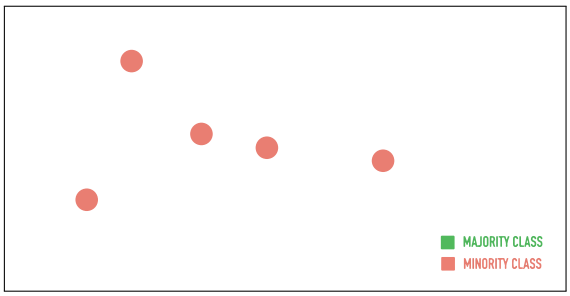


Figure 5: Example of application of One-Class Classifier

*One-Class Classifier [54].* As shown in Figure 5, a One-Class Classifier is trained only on the samples belonging to the minority class to learn the unique features of this class and accurately identify an unseen sample of this class as distinct from a sample of any other class. All instances belonging to other classes are identified as outliers. For this technique we rely on OneClassClassifier, an implementation provided by Weka [73].

| Code Smell | Min | Mean | Median | Max | Total |
|---|---|---|---|---|---|
| God Class | 0 | 5.5 | 4 | 24 | 509 |
| Spaghetti Code | 0 | 12.7 | 11 | 31 | 1443 |
| Class Data Should Be Private | 0 | 11.4 | 11 | 37 | 1150 |
| Complex Class | 0 | 6.4 | 4 | 20 | 669 |
| Long Method | 3 | 48.3 | 26 | 147 | 4763 |
| Feature Envy | 0 | 1.3 | 0 | 12 | 148 |
| Inappropriate Intimacy | 0 | 15.4 | 2 | 774 | 1788 |
| Middle Man | 0 | 0.9 | 0 | 6 | 107 |
| Refused Bequest | 0 | 6.5 | 4 | 22 | 750 |
| Speculative Generality | 0 | 9.5 | 7 | 38 | 1106 |
| Long Parameter List | 0 | 5 | 1 | 29 | 578 |

Table 3: Distribution statistics for Object-Oriented code smells

### 3.3. Subject Systems

We select software systems for which a validated dataset of code smells exists. Specifically, we relied on 125 releases of 13 open-source software systems [33]. We employed the same dataset and the same list of code smells that we used in our previous study [26] where we compare heuristics-based and Machine-Learning-based techniques for code smell detection. The dataset is available in our online appendix [29]. The systems are heterogeneous since they have different sizes, lifetimes, and belong to different application domains. The main characteristics of the considered systems are reported in the online appendix [29], as well as in the previous study [26]. Note that the dataset consists of *manually validated* code smells instances (i.e., 8,534). The distribution of code smells in the dataset is reported in Table 3. The low median number of code smells in each considered release demonstrates that code smell detection is a highly imbalanced problem.

### 3.4. Model Building and Evaluation

For each model we apply a Feature Selection step by using Correlation-based Feature Selection (CFS) [76] to remove highly correlated independent variables. Then, we tune the hyper-parameters of the classifier by applying the Grid Search algorithm [77], therefore resulting in five models that only differ for the choice of the data balancing technique to adopt.

As *independent variables* we consider the code metrics related to the structural characteristics of the software instances (e.g., size, complexity). We exploit the set of metrics originally adopted by Moha et al. [18]. In particular, given the smell detection rule, we design a model

where we employ as independent variables only the metrics used in the detection rule. For example, for *God Class* the detection rule is $ELOC > 500 \wedge (WMC + NOA) > 20 \wedge LCOM > 350$. Therefore, we train the model on the effective number of lines of code (i.e., $ELOC$), the weighted methods per class (i.e., $WMC$), the number of attributes (i.e., $NOA$), and the lack of cohesion per class (i.e., $LCOM$). Table 1 reports the features used to detect each smell, while the complete list, including the full name of the metrics, is depicted in Table 2.

Since we are interested in detecting code smells, we set the presence/absence of a specific code smell as *dependent variable* of the Machine-Learning model. This information was already available in the considered dataset.

To assess the capabilities of each of the five resulting Machine-Learning models, we adopt 10-Fold Cross Validation [78]. This methodology randomly partitions the data into 10 folds of equal size, applying a stratified sampling (e.g., each fold has the same proportion of code smell instances). A single fold is used as the test set, while the remaining ones are used as the training set. The process is repeated 10 times, using each time a different fold as the test set. For each software system and data balancing technique, we build a machine-learning model (i.e., within-project classification). The result consists of a confusion matrix for each code smell type, for each of the 125 project releases and each experimented classifier. Later, these matrices have been analysed to measure the evaluation metrics described in the following parts of the section.

To assess the effectiveness of the experimented detection techniques we compute four well-known metrics [79, 80], namely, *precision*, *recall*, *F-Measure*, and *Matthews Correlation Coefficient (MCC)*. We chose to discuss results only in terms of MCC because this metric provides a better overview with respect to the other metrics by considering all the confusion matrix [81]. The results for all the other metrics are reported in our online appendix [29].

Since we consider several datasets, we need to aggregate the results achieved to have a more precise overview of the quality of results [82]. This step has been performed in a two-fold manner (i) by aggregating the confusion matrices and (ii) by plotting the results as boxplots. Boxplots are very useful to describe the distribution of the results and provide preliminary outcomes on the comparison of different techniques. However, to draw more reliable conclusions, they need to be complemented with statistical tests. Therefore, we use the Nemenyi test [83] for statistical significance and report its results by mean of MCB (Multiple comparisons with the best) plots [84]. The elements plotted above the gray band are statistically larger than the others.

### *3.5. Results of the Study*

For each code smell, we first discuss the results by displaying boxplots, and then we evaluate their statistical significance relying on the results provided by the Nemenyi test. Note that we discarded all the cases in which at least one technique fails. The reasons behind these failures are discussed in Section 5.

Figure 6 reports the boxplots for the MCC values obtained by applying different balancing techniques. The results of the Nemenyi test, for the statistical significance, are shown in Figure 7.

The first aspect we can observe is that, regardless of the balancing technique and the code smell under analysis, MCC values are between 0 and 0,5 which indicates that Machine Learning has limited accuracy for Object-Oriented code smell detection.

The results show that *SMOTE* is the most effective technique. However, in 7 out of 11 cases, none of the balancing techniques is significantly better in terms of MCC. *No-balancing* provides good accuracy as well, since it appears six times in the group containing the most effective techniques.

An important aspect to remark is that for 4 out of 11 object-oriented code smells, *SMOTE* and *No-balancing* MCCs are significantly higher than all the other balancing techniques. This is the case of two class-level code smells (*God Class*, and *Complex Class*) and two method-level code smells (*Long Method*, and *Feature Envy*). *God Class* and *Complex Class* are the easiest class-level code smells to identify. Their detection rules are straightforward and based on easy-to-calculate metrics (e.g., size, complexity), leading to a median MCC close to 0.5 regardless of the data balancing applied. As for *Long Method* and *Feature Envy*, these are method-level smells; hence, the total number of instances to predict is much higher. We could deem *SMOTE* and *No-balancing* to have higher effectiveness where the detection metrics are simple or a large number of instances is available. However, *Long-ParameterList* is an exception. Indeed, although it is a method-level smell, the best MCC values are achieved by *One-Class Classifier*. In this specific case, *SMOTE* and *No-balancing* accuracy is slightly lower than *One-Class Classifier* but still better than all the other techniques.

Two unusual cases for which a specific discussion is required are *Middle Man* and *Speculative Generality*. *Middle Man* represents one of the rare cases in which data balancing techniques improve ML effectiveness. Indeed, *No Balancing* is the least accurate technique, with a quite clear difference to the others. As for *Speculative Generality*, results show that, regardless of the adopted data balancing technique, MCC values are very low proving that Machine-Learning is still not applicable for the detection of this smell with the set of metrics used in our study.

By and large, results suggest that there is no balancing technique which is better than the others. Indeed, different balancing techniques could be more suitable for different types of code smells. However, the highest accuracy is achieved by *No-balancing* and *SMOTE*, except for some code smells in which *One-Class Classifier* shows a higher MCC.

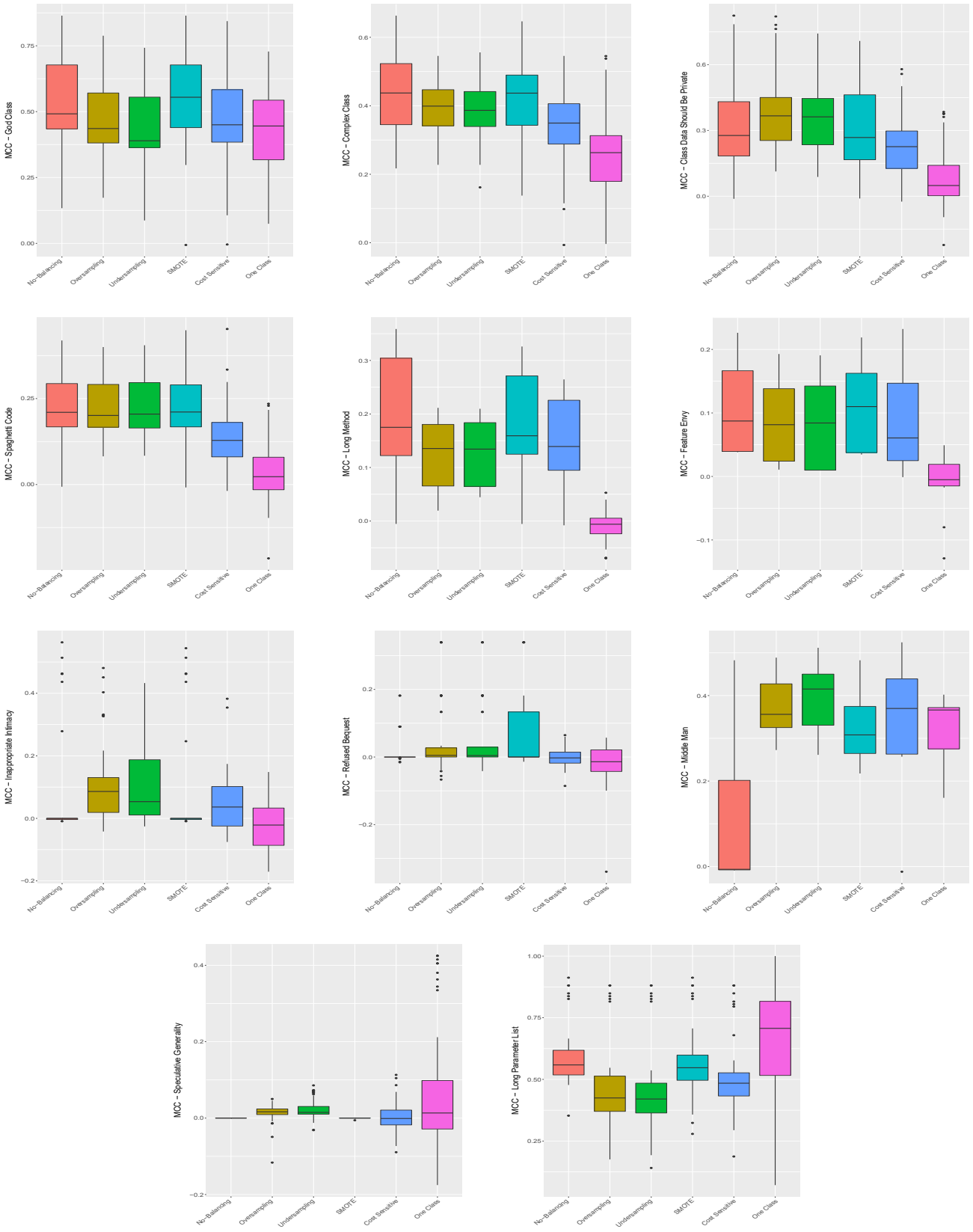


Figure 6: Boxplots representing the MCC values obtained by Naive Bayesian trained applying different balancing strategies for Object-Oriented code smells detection.

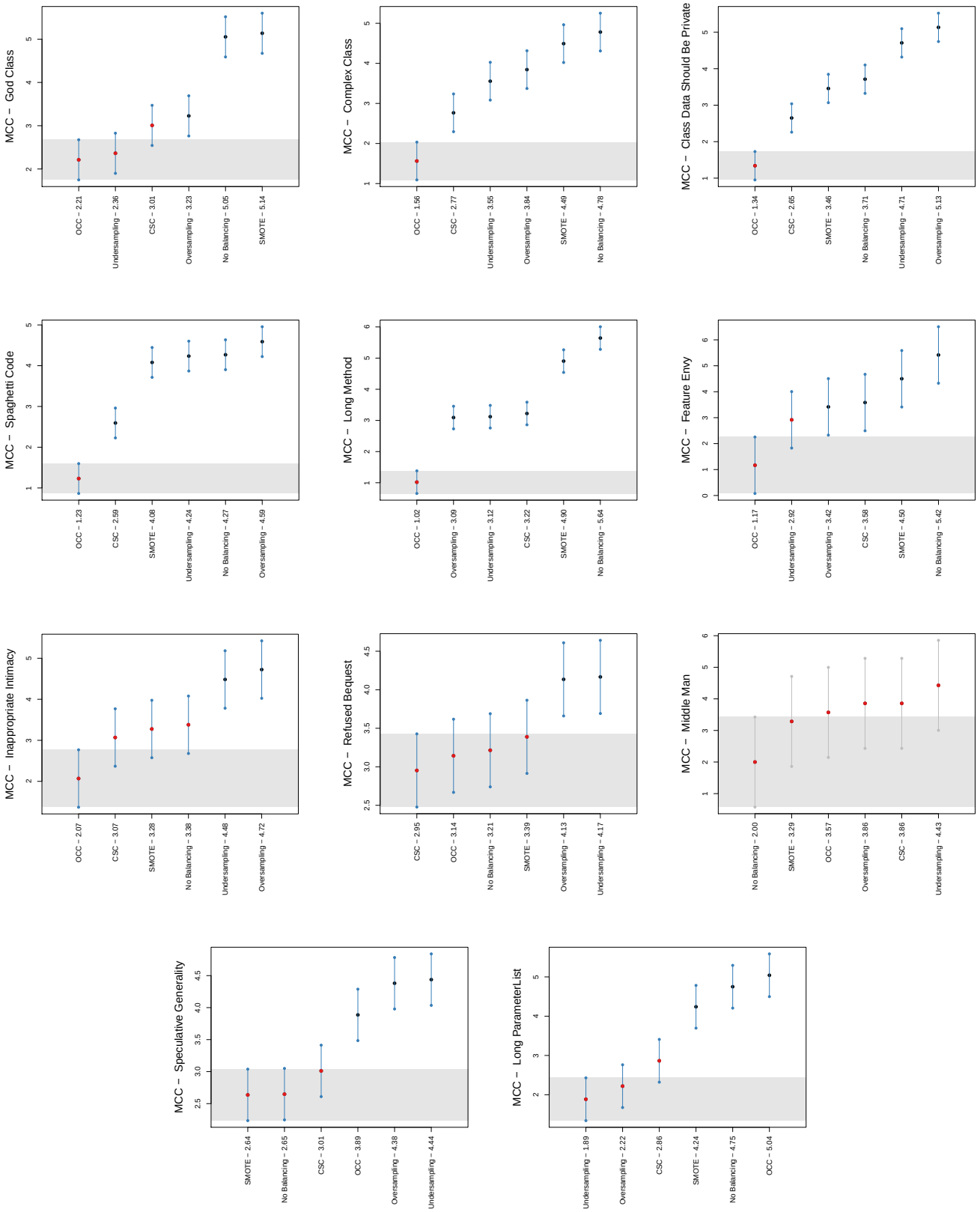


Figure 7: Plots representing the results of Nemenyi test for statistical significance between the MCC values obtained by Naive Bayesian trained applying different balancing strategies for Object-Oriented code smells detection.

**Object-Oriented Code Smell Detection**

The results show that the performance of current Machine-Learning-based approaches for detecting Object-Oriented code smells is quite limited, regardless of the adopted balancing technique ($MCC < 0.50$). Overall, *SMOTE* and *No Balancing* seem to be more effective than the other techniques.

## 4. Detection of Model-View-Control Code Smells

The purpose of the second study is to understand the impact of data balancing techniques on the accuracy of Machine-Learning algorithms in detecting the design flaws from the catalogue designed by Aniche et al. [28] who defined smells specific for systems implementing the Model-View-Control pattern. Specifically, we aim at addressing the same research questions as for the Object-Oriented code smells:

**RQ3.** Do data balancing techniques improve the effectiveness of Machine-Learning algorithms in detecting code smells specific for systems implementing Model-View-Controller pattern?

**RQ4.** Which data balancing technique is the most effective at improving the effectiveness of Machine-Learning algorithms in detecting code smells specific for systems implementing Model-View-Controller pattern?

### *4.1. Code Smells*

We analyse the code smells specific to systems adopting the Model-View-Controller pattern [28]. Such a pattern is popular across many well-know frameworks (e.g., Ruby on Rails, Spring MVC, ASP.NET MVC) [28]. In particular, we consider four code smells for which we report the heuristics needed to detect them and the metrics that we used to build the Machine-Learning models in Table 4. Such metrics are fully described in Table 5.

### *4.2. Data Balancing Techniques for Machine Learning*

We experiment the same base classifier (i.e., Naive Bayes) and the same set of data balancing techniques previously used in Section 3 and described in Section 3.2.

### *4.3. Subject Systems*

We use the dataset developed by Aniche et al. [28], consisting of 120 open-source systems. We rely on this dataset because the approach used to detect the smells has been validated with expert industrial developers in software systems implemented using Spring. This widely adopted MVC framework uses stereotypes to explicitly mark classes playing different roles (e.g., Controller classes), thus facilitating identifying the role of each class. The distribution of the smells is reported in Table 6.

### *4.4. Model Building and Evaluation*

We build and evaluate the models by following the same procedure described in Section 3.4 except the *independent variables* that were extracted from the heuristics derived by Aniche et al. [28]. Table 4 reports the features used to detect each smell, while the complete list, including the full name of the metrics, is depicted in Table 5.

As for Object-Oriented code smells, we first discuss the results by analysing the boxplots and then verify their statistical significance relying on the Nemenyi test [83]. Please consider that, also in this case, we discarded all the cases in which at least one technique fails. The reasons behind these failures are discussed in Section 5.

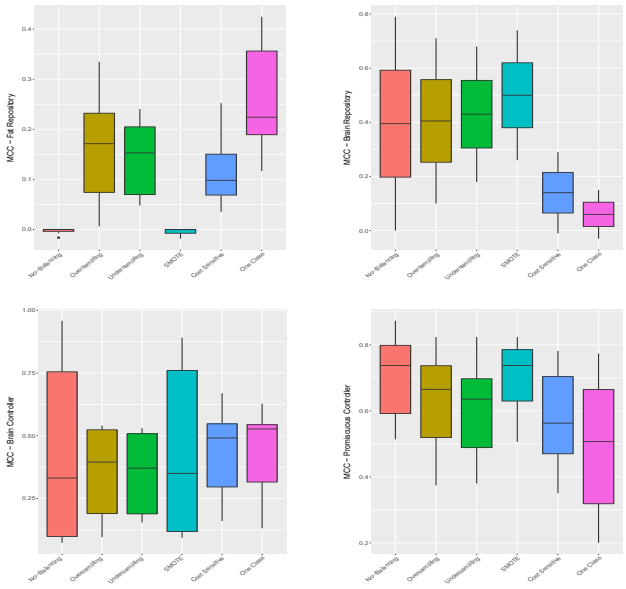


Figure 8: Boxplots representing the MCC values obtained by Naive Bayesian trained applying different balancing strategies for MVC code smells detection.

### *4.5. Results of the Study*

The results for MVC code smell detection reported in Figure 8 and Figure 9 show that ML has pretty higher accuracy when detecting this type of code smells than when detecting Object-Oriented code smells (i.e., MCC values up to $\approx 0.70$). Similarly to the Object-Oriented case, *SMOTE* and *No-balancing* show higher accuracy with respect to the other balancing techniques. As already observed, these two balancing techniques seem to be more effective where the ML algorithm has a higher prediction power. Indeed *SMOTE* achieves significantly higher MCCs in all cases except for *Fat Repository* in which MCC values are lower. Instead, *No-balancing* appears in the first group in 2 out of 4 cases. A singular case is the *Fat Repository* smell, where *One-Class Classifier* accuracy is significantly higher than the other balancing techniques. A possible motivation behind this surprising result could be found by analysing the smell distribution in Table 6.

| Code Smell | Short Description | Detection Rule | ML Model Features |
|---|---|---|---|
| Brain Repository | Repository classes that include complex business logic or queries [28] | $McCabe > \alpha \wedge SQLC > \beta$ | $McCabe$, $SQLC$ |
| Fat Repository | A Repository which deals with many Entity classes [28] | $CTE > \alpha$ | $CTE$ |
| Promiscuous Controller | Controller classes exhibiting this smell offer too many actions [28] | $NSR > \alpha \vee NSD > \beta$ | $NSR$, $NSD$ |
| Brain Controller | Controller classes with a complex control flow [28] | $NFRFC > \alpha$ | $NFRFC$ |

Table 4: Descriptions of MVC code smells along with the heuristics used to detect them and the features used by the ML models

| Acronym | Full Name | Smells |
|---|---|---|
| McCabe | McCabe's Cyclomatic Complexity | Brain Repository |
| NOR | Number of Routes | Promiscuous Controller |
| NSD | Number of Services as Dependencies | Promiscuous Controller |
| NFRFC | Non-Framework Response For a Class | Brain Controller |
| SQLC | SQL Complexity | Brain Repository |
| CTE | Calls to Entities | Fat Repository |

Table 5: Complete list of the considered metrics for the detection of Model-View-Controller code smells.

| Code Smell | Min | Mean | Median | Max | Total |
|---|---|---|---|---|---|
| Brain Repository | 0 | 0.5 | 0 | 26 | 31 |
| Fat Repository | 0 | 1.2 | 0 | 28 | 126 |
| Promiscuous Controller | 0 | 6.7 | 0 | 478 | 682 |
| Brain Controller | 0 | 1.1 | 0 | 14 | 66 |

Table 6: Distribution statistics for MVC code smells

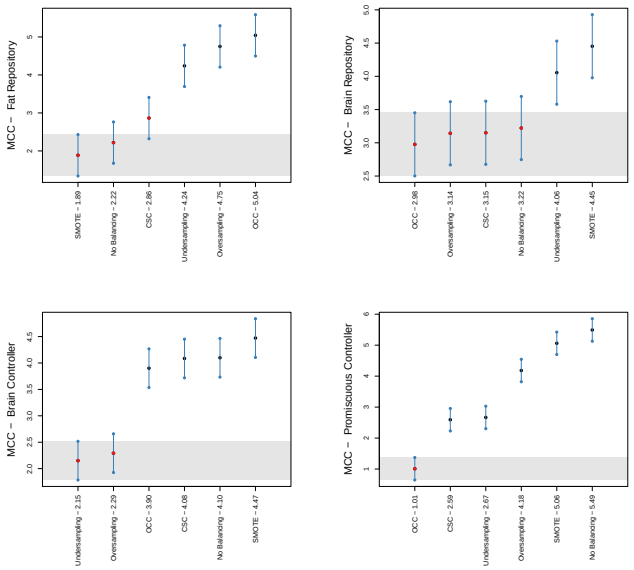


Figure 9: Plots representing the results of Nemenyi test for statistical significance between the MCC values obtained by Naive Bayesian trained applying different balancing strategies for MVC code smells detection.

Indeed, the class distribution for *Fat Repository* is almost uniform (i.e., the smelly instances are well spread across the project). Therefore, in most of the cases there are enough instances to build a reliable training set.

A final consideration is that MVC code smells are likely to be more system-dependent. Boxplots indicate a high variability of results with respect to the considered system showing very large distributions in most of the cases.

**Model-View-Controller Code Smell Detection**

With respect to the Object-Oriented case, Machine-Learning-based approaches are sharply more effective for the detection of MVC related code smells. In three out of four cases, the results are pretty good, achieving MCC values up to 0.67 and recall up to 1.00. Similarly to Object-Oriented systems, *No Balancing* and *SMOTE* are the most effective techniques.

## 5. Discussion

In this section, we discuss the results of our study. In particular, we analyse (i) the degree to which balancing techniques results are in overlap, (ii) the impact of the metrics selection on the effectiveness, and (iii) the significance of the overhead introduced by data balancing.

### *5.1. Understanding the Role of Data Balancing*

Since we described the results in quantitative terms, we provide a deeper discussion in qualitative terms. Specifically, we analyse the overlap between the results achieved by the models using different balancing technique to understand which instances they predict and whether these are complementary.

Figure 10 shows the misclassified instances obtained by the five models using the data balancing techniques and the *No-balancing* model. The axes represent the features of the model: the *x-axis* is $ELOC$, while *y-axis* is $NMNOPARAM$. '+' data points represent false negatives, while 'x' data points represent false positives. We describe this case because it nicely explains the behaviours of the balancing techniques. The model is built with two features, thus making it easier to analyse than models

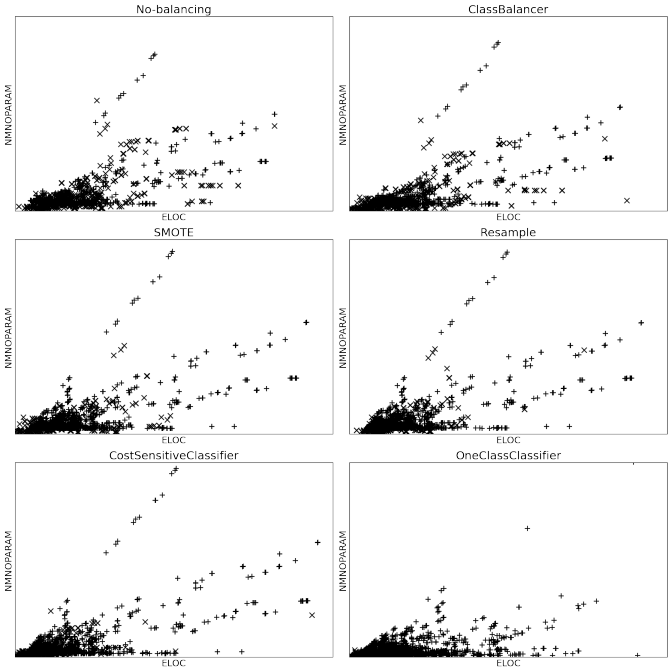


Figure 10: Scatterplots representing the misclassified instances obtained by Naive Bayesian trained applying different balancing strategies for *Spaghetti Code*. '+' data points are false negatives, while 'x' are false positives.

| Code Smell | No Smells | | SMOTE Failures | | Number of Systems |
|---|---|---|---|---|---|
| | # | % | # | % | |
| God Class | 20 | 16 | 46 | 37 | 125 |
| Spaghetti Code | 7 | 6 | 16 | 13 | 125 |
| Class Data Should Be Private | 10 | 8 | 11 | 9 | 125 |
| Complex Class | 30 | 24 | 52 | 42 | 125 |
| Long Method | 68 | 54 | 89 | 71 | 125 |
| Feature Envy | 82 | 66 | 96 | 77 | 125 |
| Inappropriate Intimacy | 3 | 2 | 72 | 58 | 125 |
| Middle Man | 64 | 51 | 104 | 83 | 125 |
| Refused Bequest | 34 | 27 | 46 | 37 | 125 |
| Speculative Generality | 3 | 2 | 11 | 9 | 125 |
| Long Parameter List | 54 | 22 | 62 | 50 | 125 |
| Brain Repository | 90 | 75 | 95 | 79 | 120 |
| Fat Repository | 87 | 72 | 94 | 78 | 120 |
| Promiscuous Controller | 44 | 37 | 73 | 61 | 120 |
| Brain Controller | 52 | 43 | 75 | 62 | 120 |

Table 7: Number of software systems not exhibiting instances of each smell (i.e., No Smells), along with the instances on which SMOTE could not be executed because of lacking instances for that specific smell (i.e., SMOTE Failures).

trained with many more features. We report the other plots as part of the online appendix [29].

The results confirm what we previously reported in Figure 6 and Figure 7. In particular, *One-Class Classifier* exhibited a high level of recall but a very low precision. This means that for code smell detection training only on the instances belonging to the minority class is not effective because these few instances poorly represent the smelly classes. A similar result is obtained for *Cost-Sensitive Classifier* which had poor precision. In particular, we notice that many points were misclassified as true, even if they were false (i.e., false negative). We can argue that giving higher weights to the instances belonging to the minority class is not effective. When analysing at the *Oversampling* and *Undersampling*, we observe that their accuracy is similar to that obtained by *No-balancing*. Therefore, we deem that these techniques are ineffective but do not worsen the accuracy achieved by the model trained without balancing. Finally, we note that *SMOTE* can slightly improve accuracy. However, it is worth remarking that some balancing techniques can fail to balance the dataset when the number of smelly instances is minimal. We tuned *SMOTE* to rely on the minimum number of smelly neighbour instances (i.e., two). If these are not available, then the algorithm fails, representing a clear disadvantage with respect to the other techniques.

Table 7 reports the number and the percentage of failures for each of the code smells under analysis. While for some code smells, there is a minimal number of failures (e.g., *Speculative Generality*), there are also smells in which the analysis fails in the majority of cases. As an example, let us consider the case of *Fat Repository*. This is one of the less frequent code smells, as also reported in Table 6: indeed, in 72% of cases, all data balancing fails due to the total absence of smelly instances in the considered system. As for *SMOTE*, it fails in 78% of cases (i.e., 72% with no smelly instances and 6% with not enough neighbours).

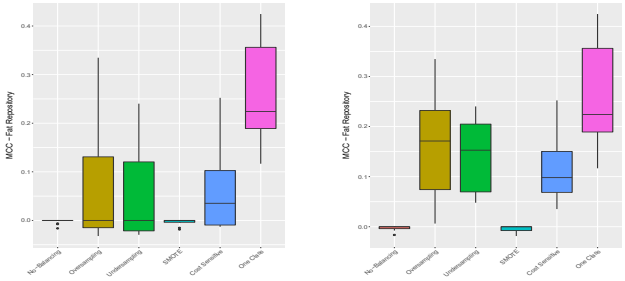

Figure 11: Boxplots representing the MCC values obtained by Naive Bayesian trained with different balancing strategies for the detection of *Fat Repository* code smell. The picture on the left includes the cases in which not all the algorithms could be executed. These cases are filtered out in the picture on the right.

Due to these failures, our analyses have been performed on a smaller population for some code smells. To avoid threatening the significance of results, we conducted a further evaluation in which we included all the systems, regardless of the failures. Figure 11 reports an example for *Fat Repository*–all the other figures are part of our online appendix [29]. The boxplot including all cases is reported on the left of the figure, while on the right side is reported the one excluding failures (i.e., the one already reported in Section 4.5). Generally, there are small differences in terms of accuracy. Indeed, for both cases, *One Class Classifier* is the most effective data balancing technique. We observed similar results for the other code smells whose boxplots are available in our online appendix [29].

Overall the results obtained on the different models show that there are no sensible differences in applying or not balancing techniques. This result suggests that tuning data balancing techniques could not be an adequate solution for code smell detection with respect to what achieved in other contexts such as defect prediction [85]. This aspect raises several issues about the feasibility of current Machine-Learning-based approaches. We deem that the meagre number of instances from the minority class (i.e., smelly instances) is the cause of this low effectiveness.

**Understanding the Role of Data Balancing**

Generally, data balancing does not significantly improve the effectiveness of Machine-Learning models for code smell detection. Training only on the instances belonging to the minority class or giving them more weight (i.e., as done by *One-Class Classifier* and *Cost-Sensitive Classifier*) is not effective because these few instances poorly represent the minority class. Resampling techniques such as *Oversampling* and *Undersampling* are ineffective but do not worsen the accuracy achieved by the model trained without balancing. Finally, *SMOTE* slightly improves the results, but in case of extremely imbalanced datasets, the training phase fails.

### *5.2. Analysing the Impact of Metric Selection*

Since Machine-Learning models achieve good accuracy only for some code smells, regardless of the adopted balancing technique, we assessed the effectiveness of the heuristic-based techniques. Our main goal is to investigate whether the low accuracy is due to the Machine-Learning techniques or caused by the reduced prediction power of the used metrics. We hypothesise that metrics with low prediction power are detrimental for both Machine Learning-based and heuristics-based approaches. This analysis was conducted only for Object-Oriented code smells where the accuracy of Machine-Learning techniques is low.

Table 8 reports the aggregate results of the evaluation metrics for (i) the Machine-Learning-based technique executed with the best balancing technique for each code smell (ML); (ii) the heuristic-based approach based on the detection rules described in Table 1 (H).

MCC values are generally low for any of the considered code smells (lower than 0.5). Except for *Long Method*, recall is always much higher than precision for heuristics-based approaches as well as for Machine-Learning-based ones. In other words, they tend to produce a large number of false positives when these metrics are employed. Therefore, such metrics might not be adequate to discriminate smelly or non-smelly instances. Comparing these results with the ones reported in Section 3.5, we note that heuristics do not outperform Machine-Learning. On the contrary, for six of the eleven object-oriented code smells, Machine-Learning-based approaches have a higher MCC. For instance, let us consider the case of *Long Parameter List* in which Machine Learning shows MCC equal to 0.58 that is much higher than the one of the heuristics-based approach (i.e., 0.12). To sum up, the results indicate that the employed set of metrics (i.e., structural metrics) are not adequate in most of the cases.

**Analysing the Impact of Metric Selection**

The results indicate that structural metrics alone are not adequate for code smell detection. This confirms previous work that deems as necessary textual and historical metrics as well as their combination with structural metrics to achieve better accuracy.

### *5.3. Inspecting the Overhead caused by Data Balancing*

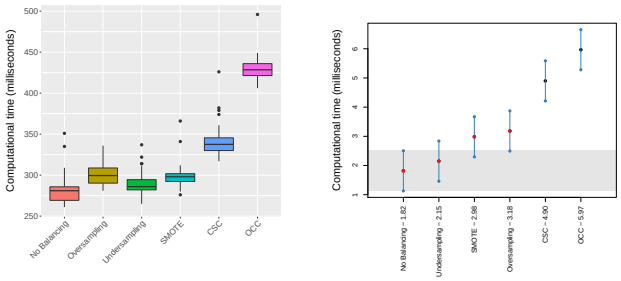


Figure 12: Boxplots and Nemenyi results representing the execution time of the different data balancing techniques.

In the previous sections, we found that some data balancing techniques improve to a limited extent the quality of results provided by Machine Learning algorithms. Given that data balancing is an additional pre-processing step in ML classification, we conducted further analysis to investigate the overhead in terms of time consumption to apply this step. Specifically, we compared the training time of the models configured with different balancing techniques. We performed 30 independent runs training the models on the most extensive system in our dataset, i.e., Eclipse 5.2.1. The selection is motivated by a twofold reason. On the one hand, having a higher number of instances should avoid (or at least reduce) failures. On the other hand, a higher number of instances led to longer execution time for all the techniques, and this may allow us to study the overhead better. We considered only one code smell (i.e., God Class). However, we believe that the results for the other smells should not be very different.

The results in Figure 12 highlight that *One-Class Classifier* and *Cost-Sensitive Classifier* take much more time than the others. This could be due to the difficulties in carrying out the training phase using a limited number of instances of the minority class. Also for the other techniques results show an overhead, although less significant

| | Code Smells Detection Comparison | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Precision | | Recall | | F-measure | | MCC | |
| | ML | H | ML | H | ML | H | ML | H |
| God Class | **0.26** | 0.08 | 0.93 | **1.00** | **0.41** | 0.16 | **0.49** | 0.28 |
| Complex Class | **0.26** | 0.23 | 0.65 | **0.72** | **0.37** | 0.35 | **0.40** | 0.37 |
| Class Data Should Be Private | **0.23** | **0.23** | **0.55** | 0.42 | **0.33** | 0.30 | **0.35** | 0.31 |
| Spaghetti Code | **0.16** | 0.11 | 0.34 | **0.47** | **0.22** | 0.18 | **0.22** | **0.22** |
| Long Method | 0.15 | **0.57** | **0.56** | 0.37 | 0.23 | **0.44** | 0.30 | **0.42** |
| Feature Envy | 0.03 | **0.05** | 0.44 | **0.46** | 0.05 | **0.10** | 0.11 | **0.15** |
| Inappropriate Intimacy | **0.27** | 0.04 | 0.15 | **0.43** | **0.19** | 0.07 | **0.19** | 0.12 |
| Middle Man | **0.16** | 0.04 | **0.87** | 0.43 | **0.28** | 0.07 | **0.37** | 0.12 |
| Refused Bequest | **0.12** | 0.04 | 0.05 | **0.40** | **0.07** | **0.07** | 0.07 | **0.11** |
| Speculative Generality | 0.01 | **0.04** | **0.65** | 0.43 | 0.02 | **0.08** | 0.02 | **0.13** |
| Long Parameter List | **0.35** | 0.04 | **0.95** | 0.41 | **0.51** | 0.08 | **0.58** | 0.12 |

Table 8: Aggregate Results for Heuristics-based and Machine-Learning-based Code Smells Detection

than the two mentioned above. In particular *Undersampling* performance is very close to the *No-balancing* one.

**Inspecting the Overhead of Data Balancing**

Except *Undersampling*, all data balancing techniques introduce significant overhead in time consumption of ML algorithms. While the two techniques based on meta-classification (i.e., *One-Class Classifier* and *Cost-Sensitive Classifier*) take much more execution time, the other ones (i.e., *Oversampling*, *Undersampling*, and *SMOTE*) show performance pretty close to *No-balancing*.

### *5.4. Implication of the Findings*

The results have implications for both researchers and practitioners. Both are interested in understanding quantitatively the effectiveness and efficiency of applying data balancing to Machine Learning code smell detectors. Furthermore, the formers are concerned about the qualitative perspective of the results. We report the implications related to (i) effectiveness, (ii) efficiency, (iii) adopted metrics, (iv) relation with previous work in different contexts.

*Effectiveness of Data Balancing for Code Smell Detection.* For Object-Oriented code smells, the results of RQ1 show that the accuracy of Machine-Learning models is quite limited (i.e., $MCC \leq 0.50$). Applying data balancing does not guarantee significantly better accuracy. Overall, *SMOTE* is the best data balancing technique among the ones that we experimented. However, if the dataset is exceptionally imbalanced, this technique fails, and data balancing should be avoided. In other cases, we did not find any statistically significant improvement in applying this technique with respect to not applying data balancing at all. For Model-View-Controller code smells, the accuracy of Machine-Learning techniques is quite good (i.e., $MCC \leq 0.67$ and $recall \leq 1.00$). However, also in this case, we did not find any statistical difference between models that apply data balancing (including *SMOTE*) and models that do not.

*Efficiency of Data Balancing for Code Smell Detection.* Introducing data balancing in a Machine Learning pipeline for code smell detection adds significant overhead in terms of the amount of time needed to train the models. However, this upkeep is only in the training phase, and it is negligible in absolute terms.

*On the Usage of Only Structural Metrics for Code Smell Detection.* On the one hand, the results for the detection of most Object-Oriented code smells confirm previous work [30, 31] that deem as necessary textual and historical metrics as well as their combination with structural metrics to achieve better accuracy. On the other hand, for the code smells specific for Model-View-Controller architectures, the results obtained adopting only structural metrics are interesting. In sum, more research should be conducted to (i) verify to what extent existing metrics for code smell detection are suitable for Machine-Learning-based models, and (ii) develop new metrics able to better characterise Object-Oriented code smells.

*Understanding the Role of Data Balancing.* Our work confirms the results obtained in bioinformatics by Dittman et al. [49] who showed that the improvements in terms of accuracy achieved by data balancing techniques are in most of the cases not statistically significant. For code smell detection, we observed that although *SMOTE* allows the model to be more accurate, in many cases, it is not applicable because of the few instances belonging to the minority class. Surprisingly but for the same reason, poor accuracy was obtained when training the models with *One-Class Classifier* [54], a technique that was designed for scenarios, such as code smell detection, where not enough counter-examples are available.

## 6. Threats to Validity

Possible threats to validity could affect the relationship between theory and observation (i.e., Construct Validity), the relationship between cause and effect (i.e., Internal Validity), the generalisability of the findings (i.e., External Validity), and the relationship between treatment and outcome (i.e., Conclusion Validity).

*Construct Validity.* The dataset choice is a threat. To analyse code smells for Object-Oriented systems, we relied on a dataset from a previous study [26] that was created considering several factors such as heterogeneity. Although the dataset has been manually-validated, we have to consider that it may be incomplete as well as imprecise. When analysing code smells for Model-View-Controller systems, we adopted a publicly available dataset [28] that was validated as well. Another threat is the construction of the machine-learning models, for which we took several aspects into account that could have possibly influenced the study, i.e., which features to consider, how to train the classifier, etc. However, the procedures followed in this respect are precise enough to ensure the validity of the study.

*Internal Validity.* The results we discussed are characterised by a great variability with respect to the smell under analysis. A possible reason could be the metric selection for code smell detection. Indeed, some of the selected metrics could represent a confounding factor threatening the internal validity of the study. To mitigate this threat, we relied on previously defined and validated metrics.

*External Validity.* For Object-Oriented systems, we considered a large dataset consisting of 125 releases of 13 open source systems belonging to different application domains and having different characteristics. A similarly heterogeneous dataset composed of 120 open-source Spring projects has been used when detecting code smells for Model-View-Controller systems. We selected Spring, a widely adopted MVC framework, because it uses stereotypes to explicitly mark classes playing the different roles (e.g., Controllers), thus making easier identifying the role of each class. As for the code smells, we selected 11 smells for Object-Oriented systems and four code smells for MVC systems that represent a large variety of design issues (e.g., smells related to complexity or excessive coupling between objects). Having a look at the ML-based algorithms, we selected Naive Bayes because in our previous study [26] it outperformed the other algorithms. Overall, the choice of this technique could be a possible threat to validity. However, this classifier was adopted in a previous study [49] that analysed the role of data balancing in bioinformatics. In that experiment, although several classifiers were adopted, the results in terms of data balancing technique to apply remained uniform across the classifiers.

Although this study shows that the choice of the classifier does not sensibly affect the results when applying data balancing techniques, further experiments with other classifiers in the context of code smell detection are needed to corroborate these findings.

*Conclusion Validity.* We exploited a set of widely-used metrics to evaluate the experimented techniques (i.e., precision, recall, F-measure, MCC). As for the machine learning model, a possible bias might have been due to the usage of 10-fold cross-validation. This strategy randomly partitions the set of data to create training and test sets: such randomness might have possibly led to the creation of biased training/test sets that have the consequence of under- or over-estimating the model accuracy.

## 7. Conclusion

In this paper, we have reported on a large-scale empirical comparison between six different balancing techniques for Machine-Learning-based code smell detection. The study considered eleven code smells for Object-Oriented systems and four code smells for systems implementing the Model-View-Controller pattern. For the former, we relied on a manually-validated dataset comprising 125 releases belonging to 13 open source systems. In contrast, for the latter, our dataset consisted of 120 Spring Model-View-Controller Open Source Systems.

The results suggest that Machine-Learning models relying on *SMOTE* achieve the best accuracy. However, its training phase is not always feasible in practice. Furthermore, avoiding balancing does not dramatically impact effectiveness. Techniques which perform training only on the minority class (i.e., *Cost-Sensitive Classifier* and *One Class Classifier*), and resampling techniques (i.e., *Class Balancer* and *Resample*) are both not effective. Existing data balancing techniques are therefore, inadequate for code smell detection. Furthermore, the results indicate that structural metrics alone are not adequate for code smell detection, confirming the previous work [30, 31] on the necessity of textual and historical metrics as well as their combination with structural metrics to achieve better accuracy. This hinders the feasibility of the current Machine-Learning-based approaches.

Our future work includes devising new techniques for handling data balancing as well as understanding whether textual and historical metrics can improve the accuracy of Machine-Learning-based detectors. Furthermore, we aim at assessing the combination of data balancing techniques and ensemble classifiers (i.e., Voting [86], Stacking [87], and ASCI [88]) to avoid the issues related to classifier selection.

## Acknowledgements

This project was partially supported by the Excellence of Science Project SECO-Assist (0015718F, FWO - Vlaanderen and F.R.S.-FNRS). Di Nucci acknowledges

the support of the European Commission grant no. 825040 (H2020 - RADON).

## References


[1] W. Cunningham, The wycash portfolio management system, ACM SIGPLAN OOPS Messenger 4 (2) (1993) 29–30.

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

[3] E. V. de Paulo Sobrinho, A. De Lucia, M. de Almeida Maia, A systematic literature review on bad smells—5 w's: which, when, what, who, where, IEEE Transactions on Software Engineering.

[4] M. I. Azeem, F. Palomba, L. Shi, Q. Wang, Machine learning techniques for code smell detection: A systematic literature review and meta-analysis, Information and Software Technology (2019) in press.

[5] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, D. Poshyvanyk, When and why your code starts to smell bad (and whether the smells go away), IEEE Transactions on Software Engineering.

[6] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, D. Poshyvanyk, An empirical investigation into the nature of test smells, in: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ACM, 2016, pp. 4–15.

[7] A. Chatzigeorgiou, A. Manakos, Investigating the evolution of bad smells in object-oriented code, in: Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the, IEEE, 2010, pp. 106–115.

[8] S. Olbrich, D. S. Cruzes, V. Basili, N. Zazworka, The evolution and impact of code smells: A case study of two open source systems, in: Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on, IEEE, 2009, pp. 390–400.

[9] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, A. De Lucia, A large-scale empirical study on the lifecycle of code smell co-occurrences, Information and Software Technology 99 (2018) 1–10.

[10] F. Palomba, A. Zaidman, Does refactoring of test smells induce fixing flaky tests?, in: Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on, IEEE, 2017, pp. 1–12.

[11] F. Palomba, A. Zaidman, The smell of fear: On the relation between test smells and flaky tests, Empirical Software Engineering Journal (2019) in press.

[12] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, G. Antoniol, An exploratory study of the impact of antipatterns on class change- and fault-proneness, Empirical Software Engineering 17 (3) (2012) 243–275.

[13] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, A. De Lucia, On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation, Empirical Software Engineering 23 (3) (2018) 1188–1221.

[14] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, Do they really smell bad? a study on developers' perception of bad code smells, in: Software maintenance and evolution (ICSME), 2014 IEEE international conference on, IEEE, 2014, pp. 101–110.

[15] A. Yamashita, L. Moonen, Do code smells reflect important maintainability aspects?, in: 2012 28th IEEE International Conference on Software Maintenance (ICSM), 2012, pp. 306–315.

[16] D. Taibi, A. Janes, V. Lenarduzzi, How developers perceive smells in source code: A replicated study, Information and Software Technology 92 (2017) 223–235.

[17] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, E. Figueiredo, A review-based comparative study of bad smell detection tools, in: Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering, ACM, 2016, p. 18.

[18] N. Moha, Y.-G. Gueheneuc, L. Duchien, A.-F. Le Meur, Decor: A method for the specification and detection of code and design smells, IEEE Trans. on Software Engineering 36 (1) (2010) 20–36.

[19] F. Palomba, A. De Lucia, G. Bavota, R. Oliveto, Anti-pattern detection: Methods, challenges, and open issues, in: Advances in Computers, Vol. 95, Elsevier, 2014, pp. 201–238.

[20] M. Zhang, T. Hall, N. Baddoo, Code bad smells: a review of current knowledge, Journal of Software Maintenance and Evolution: research and practice 23 (3) (2011) 179–202.

[21] F. A. Fontana, J. Dietrich, B. Walter, A. Yamashita, M. Zanoni, Antipattern and code smell false positives: Preliminary conceptualization and classification, in: Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on, Vol. 1, IEEE, 2016, pp. 609–613.

[22] M. V. Mäntylä, C. Lassenius, Subjective evaluation of software evolvability using code smells: An empirical study, Empirical Software Engineering 11 (3) (2006) 395–431.

[23] F. A. Fontana, P. Braione, M. Zanoni, Automatic detection of bad smells in code: An experimental assessment., Journal of Object Technology 11 (2) (2012) 5–1.

[24] F. A. Fontana, M. V. Mäntylä, M. Zanoni, A. Marino, Comparing and experimenting machine learning techniques for code smell detection, Empirical Software Engineering 21 (3) (2016) 1143–1191.

[25] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, A. De Lucia, Detecting code smells using machine learning techniques: are we there yet?, in: 25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER2018): REproducibility Studies and NEgative Results (RENE) Track, Institute of Electrical and Electronics Engineers (IEEE), 2018, pp. 612–621.

[26] F. Pecorelli, F. Palomba, D. Di Nucci, A. De Lucia, Comparing heuristic and machine learning approaches for metric-based code smell detection, in: Proceedings of the 27th International Conference on Program Comprehension, IEEE Press, 2019, pp. 93–104.

[27] F. Pecorelli, D. Di Nucci, C. De Roover, A. De Lucia, On the role of data balancing for machine learning-based code smell detection, in: Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation, ACM, 2019, pp. 19–24.

[28] M. Aniche, G. Bavota, C. Treude, M. A. Gerosa, A. van Deursen, Code smells for model-view-controller architectures, Empirical Software Engineering 23 (4) (2018) 2121–2157.

[29] F. Pecorelli, D. Di Nucci, C. De Roover, A. De Lucia, A large empirical assessment on the role of data balancing in machine-learning-based code smell detection - online appendix `https://figshare.com/s/5da162e21b8d54fbfce8` (2019).

[30] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, A. De Lucia, Mining version histories for detecting code smells, IEEE Transactions on Software Engineering 41 (5) (2015) 462–489.

[31] F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, A. Zaidman, A textual-based technique for smell detection, in: Program Comprehension (ICPC), 2016 IEEE 24th International Conference on, IEEE, 2016, pp. 1–10.

[32] T. J. McCabe, A complexity measure, IEEE Transactions on software Engineering (4) (1976) 308–320.

[33] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, A. De Lucia, On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation, Empirical Software Engineering (2017) 1–34.

[34] J. Kreimer, Adaptive detection of design flaws, Electronic Notes in Theoretical Computer Science 141 (4) (2005) 117–136.

[35] L. Amorim, E. Costa, N. Antunes, B. Fonseca, M. Ribeiro, Experience report: Evaluating the effectiveness of decision trees for detecting code smells, in: Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on, IEEE, 2015, pp. 261–269.

[36] S. Vaucher, F. Khomh, N. Moha, Y.-G. Guéhéneuc, Tracking design smells: Lessons from a study of god classes, in: Reverse Engineering, 2009. WCRE'09. 16th Working Conference on, IEEE, 2009, pp. 145–154.
[37] A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y.-G. Gueheneuc, E. Aimeur, Smurf: A svm-based incremental anti-pattern detection approach, in: Reverse engineering (WCRE), 2012 19th working conference on, IEEE, 2012, pp. 466–475.
[38] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, H. Sahraoui, Bdtex: A gqm-based bayesian approach for the detection of antipatterns, Journal of Systems and Software 84 (4) (2011) 559–572.
[39] S. Hassaine, F. Khomh, Y.-G. Guéhéneuc, S. Hamel, Ids: An immune-inspired approach for the detection of software design smells, in: Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the, IEEE, 2010, pp. 343–348.
[40] R. Oliveto, F. Khomh, G. Antoniol, Y.-G. Guéhéneuc, Numerical signatures of antipatterns: An approach based on b-splines, in: Software maintenance and reengineering (CSMR), 2010 14th European Conference on, IEEE, 2010, pp. 248–251.
[41] F. A. Fontana, M. Zanoni, A. Marino, M. V. Mantyla, Code smell detection: Towards a machine learning-based approach, in: Software Maintenance (ICSM), 2013 29th IEEE International Conference on, IEEE, 2013, pp. 396–399.
[42] F. A. Fontana, M. Zanoni, Code smell severity classification using machine learning techniques, Knowledge-Based Systems 128 (2017) 43–58.
[43] S. Maes, K. Tuyls, B. Vanschoenwinkel, B. Manderick, Credit card fraud detection using bayesian and neural networks, in: Proceedings of the 1st international naiso congress on neuro fuzzy technologies, 2002, pp. 261–270.
[44] T. Hall, S. Beecham, D. Bowes, D. Gray, S. Counsell, A systematic literature review on fault prediction performance in software engineering, IEEE Transactions on Software Engineering 38 (6) (2012) 1276–1304.
[45] K. Kourou, T. P. Exarchos, K. P. Exarchos, M. V. Karamouzis, D. I. Fotiadis, Machine learning applications in cancer prognosis and prediction, Computational and structural biotechnology journal 13 (2015) 8–17.
[46] G. E. Batista, R. C. Prati, M. C. Monard, A study of the behavior of several methods for balancing machine learning training data, ACM SIGKDD explorations newsletter 6 (1) (2004) 20–29.
[47] N. V. Chawla, Data mining for imbalanced datasets: An overview, in: Data mining and knowledge discovery handbook, Springer, 2009, pp. 875–886.
[48] H. He, E. A. Garcia, Learning from imbalanced data, IEEE Transactions on knowledge and data engineering 21 (9) (2009) 1263–1284.
[49] D. J. Dittman, T. M. Khoshgoftaar, A. Napolitano, Selecting the appropriate data sampling approach for imbalanced and high-dimensional bioinformatics datasets, in: 2014 IEEE International Conference on Bioinformatics and Bioengineering, IEEE, 2014, pp. 304–310.
[50] D. J. Dittman, T. M. Khoshgoftaar, R. Wald, A. Napolitano, Comparison of data sampling approaches for imbalanced bioinformatics data, in: The twenty-seventh international FLAIRS conference, 2014.
[51] G. M. Weiss, F. Provost, Learning when training data are costly: The effect of class distribution on tree induction, Journal of artificial intelligence research 19 (2003) 315–354.
[52] N. Japkowicz, Concept-learning in the presence of between-class and within-class imbalances, in: Conference of the Canadian society for computational studies of intelligence, Springer, 2001, pp. 67–77.
[53] N. V. Chawla, C4. 5 and imbalanced data sets: investigating the effect of sampling method, probabilistic estimate, and decision tree structure, in: Proceedings of the ICML, Vol. 3, 2003, p. 66.
[54] D. M. J. Tax, One-class classification: Concept learning in the absence of counter-examples.
[55] M. Galar, A. Fernandez, E. Barrenechea, H. Bustince, F. Herrera, A review on ensembles for the class imbalance problem: bagging-, boosting-, and hybrid-based approaches, IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews) 42 (4) (2011) 463–484.
[56] R. E. Schapire, The strength of weak learnability, Machine learning 5 (2) (1990) 197–227.
[57] L. Breiman, Bagging predictors, Machine learning 24 (2) (1996) 123–140.
[58] T. M. Khoshgoftaar, A. Fazelpour, D. J. Dittman, A. Napolitano, Effects of the use of boosting on classification performance of imbalanced bioinformatics datasets, in: 2014 IEEE International Conference on Bioinformatics and Bioengineering, IEEE, 2014, pp. 420–426.
[59] D. J. Dittman, T. M. Khoshgoftaar, A. Napolitano, The effect of data sampling when using random forest on imbalanced bioinformatics data, in: 2015 IEEE international conference on information reuse and integration, IEEE, 2015, pp. 457–463.
[60] D. J. Dittman, T. M. Khoshgoftaar, A. Napolitano, Is data sampling required when using random forest for classification on imbalanced bioinformatics data?, in: Theoretical Information Reuse and Integration, Springer, 2016, pp. 157–171.
[61] L. Breiman, Random forests, Machine learning 45 (1) (2001) 5–32.
[62] R. Arcoverde, A. Garcia, E. Figueiredo, Understanding the longevity of code smells: preliminary results of an explanatory survey, in: Proceedings of the International Workshop on Refactoring Tools, ACM, 2011, pp. 33–36.
[63] A. Chatzigeorgiou, A. Manakos, Investigating the evolution of bad smells in object-oriented code, in: Proceedings of the 2010 Seventh International Conference on the Quality of Information and Communications Technology, QUATIC '10, IEEE Computer Society, 2010, pp. 106–115.
[64] R. Peters, A. Zaidman, Evaluating the lifespan of code smells using software repository mining, in: European Conference on Software Maintenance and ReEngineering, IEEE, 2012, pp. 411–416.
[65] S. Olbrich, D. S. Cruzes, V. Basili, N. Zazworka, The evolution and impact of code smells: A case study of two open source systems, in: Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement, ESEM '09, 2009, pp. 390–400.
[66] M. Abbes, F. Khomh, Y.-G. Gueheneuc, G. Antoniol, An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension, in: Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering, CSMR '11, IEEE Computer Society, 2011, pp. 181–190.
[67] A. Yamashita, L. Moonen, Exploring the impact of inter-smell relations on software maintainability: An empirical study, in: International Conference on Software Engineering (ICSE), IEEE, 2013, pp. 682–691.
[68] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, G. Antoniol, An exploratory study of the impact of antipatterns on class change- and fault-proneness, Empirical Software Engineering 17 (3) (2012) 243–275.
[69] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, A. De Lucia, On the diffuseness and the impact on maintainability of code smells: A large scale empirical study, Empirical Software Engineering (2017) to appear.
[70] A. Yamashita, L. Moonen, Do code smells reflect important maintainability aspects?, in: International Conference on Software Maintenance (ICSM), IEEE, 2012, pp. 306–315.
[71] G. H. John, P. Langley, Estimating continuous distributions in bayesian classifiers, in: Proceedings of the Eleventh conference on Uncertainty in artificial intelligence, Morgan Kaufmann Publishers Inc., 1995, pp. 338–345.
[72] C. X. Ling, C. Li, Data mining for direct marketing: Problems and solutions., in: Kdd, Vol. 98, 1998, pp. 73–79.
[73] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, I. H. Witten, The weka data mining software: an update, ACM SIGKDD explorations newsletter 11 (1) (2009) 10–18.

[74] N. V. Chawla, K. W. Bowyer, L. O. Hall, W. P. Kegelmeyer, Smote: synthetic minority over-sampling technique, Journal of artificial intelligence research 16 (2002) 321–357.
[75] S. Kotsiantis, D. Kanellopoulos, P. Pintelas, et al., Handling imbalanced datasets: A review, GESTS International Transactions on Computer Science and Engineering 30 (1) (2006) 25–36.
[76] M. A. Hall, Correlation-based feature selection for machine learning, Tech. rep. (1998).
[77] J. Bergstra, Y. Bengio, Random search for hyper-parameter optimization, Journal of Machine Learning Research 13 (Feb) (2012) 281–305.
[78] M. Stone, Cross-validatory choice and assessment of statistical predictions, Journal of the royal statistical society. Series B (Methodological) (1974) 111–147.
[79] R. Baeza-Yates, B. d. A. N. Ribeiro, et al., Modern information retrieval, New York: ACM Press; Harlow, England: Addison-Wesley,, 2011.
[80] D. M. Powers, Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation.
[81] M. Shepperd, D. Bowes, T. Hall, Researcher bias: The use of machine learning in software defect prediction, IEEE Transactions on Software Engineering 40 (6) (2014) 603–616.
[82] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, E. Merlo, Recovering traceability links between code and documentation, IEEE transactions on software engineering 28 (10) (2002) 970–983.
[83] P. Nemenyi, Distribution-free multiple comparisons, in: Biometrics, Vol. 18, International Biometric Soc 1441 I ST, NW, SUITE 700, WASHINGTON, DC 20005-2210, 1962, p. 263.
[84] A. J. Koning, P. H. Franses, M. Hibon, H. O. Stekler, The m3 competition: Statistical tests of the results, International Journal of Forecasting 21 (3) (2005) 397–409.
[85] A. Agrawal, T. Menzies, Is better data better than better data miners?: on the benefits of tuning smote for defect prediction, in: Proceedings of the 40th International Conference on Software engineering, ACM, 2018, pp. 1050–1061.
[86] J. Kittler, M. Hatef, R. P. Duin, J. Matas, On combining classifiers, IEEE transactions on pattern analysis and machine intelligence 20 (3) (1998) 226–239.
[87] D. H. Wolpert, Stacked generalization, Neural networks 5 (2) (1992) 241–259.
[88] D. Di Nucci, F. Palomba, R. Oliveto, A. De Lucia, Dynamic selection of classifiers in bug prediction: An adaptive method, IEEE Transactions on Emerging Topics in Computational Intelligence 1 (3) (2017) 202–212.