Code Smell Detection and Identification in Imbalanced Environments

Sofien Boutaib[a,*], Slim Bechikh[a,*], Fabio Palomba[b,*], Maha Elarbi[a], Lamjed Ben Said[a]

[a]*SMART Lab, ISG, University of Tunis, Tunisia*
[b]*University of Salerno, Italy*

Abstract

Context. Code smells are sub-optimal design choices that could lower software maintainability. ***Objective.*** Previous literature did not consider an important characteristic of the smell detection problem, namely data imbalance. When considering a high number of code smell types, the number of smelly classes is likely to largely exceed the number of non-smelly ones, and vice versa. Moreover, most studies did address the smell identification problem, which is more likely to present a higher imbalance as the number of smelly classes is relatively much less than the number of non-smelly ones. Furthermore, an additional research gap in the literature consists in the fact that the number of smell type identification methods is very small compared to the detection ones. ***Research gap.*** The main challenges in smell detection and identification in an imbalanced environment are: (1) the structuring of the smell detector that should be able to deal with complex splitting boundaries and small disjuncts, (2) the design of the detector quality evaluation function that should take into account data imbalance, and (3) the efficient search for effective software metrics' thresholds that should well characterize the different smells. Furthermore, the number of smell type identification methods is very small compared to the detection ones. ***Method.*** We propose ADIODE, an effective search-based engine that is able to deal with all the above-described challenges not only for the smell detection case but also for the identification one. Indeed, ADIODE is an EA (Evolutionary Algorithm) that evolves a population of detectors encoded as ODTs (Oblique Decision Trees) using the F-measure as a fitness function. This allows ADIODE to efficiently approximate globally-optimal detectors with effective oblique splitting hyper-planes and metrics' thresholds. ***Results.*** A comparative experimental study on six open-source software systems demonstrates the merits and the outperformance of our approach compared to four of the most representative and prominent baseline techniques available in literature. The detection results show that the F-measure of ADIODE ranges between 91.23 % and 95.24 %, and its AUC lies between 0.9273 and 0.9573. Similarly, the identification results indicate that the F-measure of ADIODE varies between 86.26 % and 94.5 %, and its AUC is between 0.8653 and 0.9531.

Keywords: Code smells detection, Smell type identification, Imbalanced data classification, Oblique decision tree, Evolutionary algorithm.

1. Introduction

Over the lifecycle, software systems are subject to many changes that are meant to maintain high their business level (Palomba et al., 2014). Regrettably, these changes are often carried out under time pressure and thus push software developers to set aside good programming guidelines. The deadline respect requirement, called the technical debt (Cunningham, 1993), may cause the immaturity of the developed software. Code smells (a.k.a., anti-patterns) (Fowler & Beck, 1999), i.e., symptoms of poor design and implementation solutions (Palomba et al., 2018b), represent the principal factor leading to serious issues regarding the maintenance of software (Catolino et al., 2019; Vassallo et al., 2019; Palomba et al., 2018c; Palomba & Zaidman, 2019). Unfortunately, the existence of smells could deteriorate essential code quality aspects such as understandability and changeability, which could cause the introduction of faults (Yamashita & Moonen, 2013a). For instance, the code smell "*Long Parameter List*" could deteriorate: (1) *readability* since it may be difficult to read many parameters simultaneously and (2) *changeability* because time-consuming modifications could be needed especially if the concerned method is frequently called. Moreover, this smell type could also push the developer to introduce defects, since it is more likely to make mistakes when a method call contains a high number of parameters. Such code smell could be removed using one of the appropriate refactoring techniques (Yamashita, 2012). Unfortunately, this choice heavily depends on the definition of the code smells, which is still so far a challenging issue due to the subjectivity of software engineers in framing each smell definition (Mäntylä & Lassenius, 2006). A considerable number of studies have been conducted to investigate the effects of code smells on non-functional properties such as understandability (Yamashita & Moonen, 2013b) as well as change- and fault-proneness (Khomh et al., 2012; Palomba et al., 2018b; Tufano et al., 2016; Spadini et al., 2018), which are among the main requirements for software evolution (Sjoberg et al., 2013; Yamashita & Moonen, 2013c). Based on these studies, several smell detection tools have been proposed and used to detect refactoring opportunities (Palomba et al., 2015; dos Santos Neto et al., 2015; Palomba et al., 2018d, 2016, 2017; Crasso et al., 2009; Vidal & Marcos, 2012). In fact, most approaches were proposed for the detection case, while only a much reduced number of approaches have focused on the identification task (Rasool & Arshad, 2017; Liu et al., 2015).

Anti-pattern detection tools could be classified into two main categories: (1) *Rule*-based tools and (2) *Search*-based ones. The former tools use a set of predefined rules that are based on the combination of metrics and thresholds (Sharma & Spinellis, 2018). Unfortunately, such kind of detectors suffers from the problem of threshold calibration; which has been reported to push the software developers to ignore the outputted anti-patterns (Di Nucci et al., 2018). It is worth noting that the threshold calibration difficulty dramatically increases with the number of the considered smell types. The latter

*Corresponding author

Email addresses: boutaibsofien@yahoo.fr (Sofien Boutaib), slim.bechikh@fsegn.rnu.tn (Slim Bechikh), fpalomba@unisa.it (Fabio Palomba), arbi.maha@yahoo.com (Maha Elarbi), lamjed.bensaid@isg.rnu.tn (Lamjed Ben Said)

tools, i.e., search-based ones, do not face such difficulties as they have the ability to automatically calibrate the different thresholds based on a base of good and/or bad pattern examples (Mansoor et al., 2017). Such detectors have demonstrated very interesting results thanks not only to the adaptive threshold tuning but also to their capacity to escape local optima; and hence providing the user with a set of *globally-optimal* detection rules as possible (Ouni et al., 2013). Despite their ability to automatically calibrate metrics' thresholds through the evolutionary optimization process, the search-based approaches could generate ineffective (meaningless) thresholds' values that negatively influence not only the effectiveness but also the efficiency of the search process. This serious issue will be detailed and then solved in Section 2.2.1 as illustrated later by Figure 4. It is important to note that, in some cases, the software engineer would like to interact with the system to specify his/her thresholds' values based on his/her knowledge and experience. The interaction is just optional in ADIODE and it is the software engineer who decides the interaction instant(s) (e.g., after a user-specified number of generations of the evolutionary process). During the interaction, the user is assisted with the set of effective thresholds for each node and for each tree. It is up to him/her to decide to: (1) either choose an already generated threshold by the Kretowski-&-Grzes method or (2) define a different threshold based on not only his/her expertise but also the set of already generated effective thresholds that may assist him/her. This means that there are no specific circumstances under which the user chooses one of the two alternatives. Eventually, once the interaction is terminated, ADIODE continues its execution using its default effective threshold definition strategy.

Unfortunately, almost all the available tools and works did not consider the data imbalance problem (Haixiang et al., 2017). Table 9 shows that all existing approaches have not consider the data imbalance problem in their algorithmic behaviors. In fact, the smell detection problem corresponds to a binary classification problem where there are two data classes (we use the term "data class" to denote a set of instances, in order to avoid confusion with the term "software class"): (1) the *majority* data class and (2) the *minority* one. When the number of considered smell types is low, e.g., a detector considers the detection on only Blob classes, it is more likely that the minority data class corresponds to the one containing smelly software classes, and vice versa. The fact that the cardinality of the minority data class is usually much less than the cardinality of the majority one (Haixiang et al., 2017; Obregon et al., 2019; Devarriya et al., 2020) could significantly deteriorate the performance of the detection tools. The imbalance issue could be even higher in the smell type identification problem, which corresponds to a detection problem while considering only a unique type of anti-patterns. This could be explained by the increase of the imbalance ratio (Pecorelli et al., 2019a), which is defined in the identification task as the ratio of the number of smelly classes to the number of non-smelly ones. In summary, most existing works (Rasool & Arshad, 2017; Sharma & Spinellis, 2018) do not consider the data imbalance issue, which depends on the number of considered smell types. Besides, the number of works dedicated for smell type identification, which is usually a more imbalanced problem, is very small.

Motivated by these observations, we consider, in this paper, the code smells detection problem as an imbalanced binary data classification problem. To solve this latter, we

propose ADIODE (**A**nti-pattern **D**etection and **I**dentification using **O**blique **D**ecision tree **E**volution) as a new method to detect and/or identify code smells. Our ADIODE takes as input a base of anti-pattern examples and then evolves a set of ODTs (Oblique Decision Trees) using an EA (Evolutionary Algorithm). Our choice is justified by:

1. The performance of ODTs in imbalanced data classification through the machine learning literature could be explained by the use of oblique splitting hyper-planes in addition to orthogonal ones (Murthy et al., 1994; Wickramarachchi et al., 2016; Das et al., 2018) as detailed in Appendix C;

2. The ability of EAs to escape local optima in the ODT search space (Barros et al., 2012), which is not the case of greedy machine learning algorithms such as OC1 (Murthy et al., 1993) and CART-LC (Breiman et al., 1984) that usually output a locally-optimal classifier (Please, refer to Appendix D);

3. The good structuring of the smell detectors as ODTs; which is not the case of existing search-based detection approaches that evolves a set of ad-hoc rules; and

4. The ability of ADIODE to be used with one or several smell types; which makes it useful for both cases: detection (i.e., considering simultaneously many smell types) and identification (i.e., considering a single smell type). As ADIODE uses two main concepts from the artificial intelligence field, namely the evolutionary computation and the ODT (i.e., a machine learning concept), it could be seen as an intelligent expert system (Hawes, 2011) that evolves a set of IF-THEN rules (encoded as ODTs) using a base of labelled smell examples (knowledge base), with the aim to support the software engineers in the detection and identification of code smells. Appendices C and D details the motivations behind the use of these artificial intelligence concepts.

We evaluated the performance of ADIODE in an empirical study involving six open-source systems, comparing it with four state-of-the-art baseline approaches. The results of our study clearly highlight that the proposed solution has better detection and identification accuracy with respect to all the considered baselines.

To sum up, the main contribution of this paper could be summarized as follows:

1. Demonstrating that smell detection (and also identification) corresponds to an imbalanced binary data classification problem;

2. Proposing ADIODE as a new method and tool to detect and identify code smells;

3. Demonstrating the ability of ADIODE in dealing with data imbalance thanks to its three main distinctions: (a) the solution encoding as ODT, (b) the use of an insensitive metric to data imbalance in binary classification that is the F-measure as a fitness function, and (c) the use of Kretowski-&-Grzes method to find effective thresholds for the efficacious definition of oblique splits of ODTs;

4. Showing the performance of our ADIODE method on a set of detailed and statistically analyzed comparative experiments on six commonly-used open source projects with respect to four existing recent and prominent works.

Structure of the paper. Section 2 presents the main motivations behind our work and describes in detail our proposed approach ADIODE. Section 3 reports the comparative experimental results with respect to the state-of-the-art approaches, while Section 4 discusses the main findings of the study as well as the implications of our work for researchers and practitioners. In Section 5, we discuss the different threats that could affect the validity of our experimentations. Section 6 summarizes the state-of-the-art of code smells detection. Finally, Section 7 concludes the paper and gives some avenues for future research.

2. Proposed approach: ADIODE

This section is devoted to describe and detail our ADIODE approach for code smells detection and identification. It is worth noting that both tasks are performed using the same mechanism, but using different bases of examples. For the case of detection, this base could contain a high number of smells or the opposite; while for the case of identification, the base contains just a single smell type. We conclude that identification is a special case of detection where the tool works only on a single anti-pattern type. To ease the understanding of our approach, we first give the main motivations behind the design of our ADIODE method. Second, we illustrate the global schema of our detection method. Third, we detail how the detector, which corresponds to an ODT, is encoded and then decoded within the EA. Fourth, we show how the population of detectors (ODTs) is varied and optimized using the EA (i.e., a Genetic Algorithm (GA)) and this is done by means of the fitness evaluation, selection, crossover and mutation modules. Finally, we describe how the generated detectors by the EA could be used for detection and identification using the ensemble machine learning technique of majority voting.

2.1. Main ideas and motivations

Motivated by all these weaknesses, our proposed ADIODE method optimizes a set of ODTs, where each one of them is evaluated using the *AUC* (Area Under Curve) metric (Baeza-Yates et al., 1999) with respect to a base of smell instances. The main merits of our approach are detailed next. First, we adopt the ODT as solution representation (i.e., detector encoding) in ADIODE since the ODT is able to generate any kind of decision boundaries: oblique and axis-parallel. This is thanks to the node representation in ODTs, which corresponds to a weighted combination of features, where features are the used software metrics. This detector encoding allows ADIODE to be more fitted to detect minority instances, including small disjuncts (Das et al., 2018) (Please, refer to Appendix C for details); which is not the case of existing search-based approaches. Second, the fitness measures used by almost all search-based tools are not well-suited to imbalanced classification, which may introduce a significant bias in their obtained results. For this

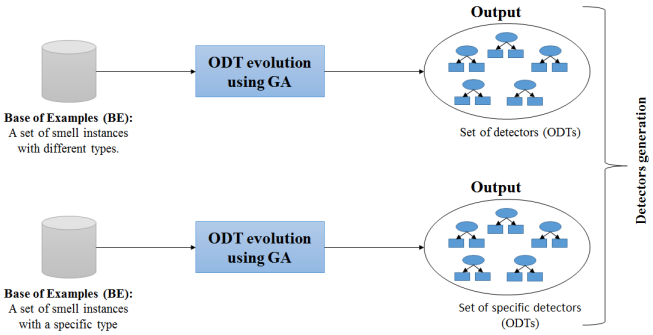
reason, we have chosen the *AUC* as a fitness function in ADIODE since it is reported that it is an effective metric in the case of class imbalance through the machine learning literature. In regard to rule-based tools, our ADIODE method has the advantage of finding automatically optimized values for the software metrics' thresholds. Finally, we should note that our approach could output specific detectors for each smell type when the BE contains a single type smells. All these motivating characteristics of ADIODE will be next discussed and shown in the comparative experimental study.

2.1.1. Main Schema

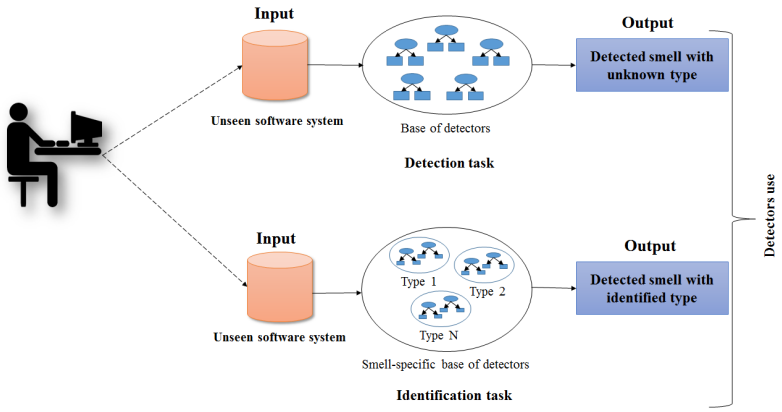
As illustrated by Figure 1, the ADIODE method is composed of two major modules: (1) The smell detectors generation module and (2) The smell detectors application module. The first one outputs a set of optimized ODTs, while the second one applies these ODTs to detect or identify the code smells on an unseen software system (i.e., a set of unseen software classes). It is important to mention that, in the detection process, all detectors are merged together into a single base of detectors. However, for the identification case, ADIODE outputs a set of specialized detectors that were trained on the considered smell type. In this way, we obtain a base of detectors for each smell type (Blob, Data Class, Feature Envy, etc.). Thus, once the user would like to analyze an unseen software class, the latter is examined with the specific detectors using the majority voting strategy to decide the smell types it contains. This process is further detailed later in Section 2.3.

2.1.2. Individual encoding

To ease the ODT representation, we have chosen to use a two-array encoding with breadth-first order as recommended by (Jankowski & Jackowski, 2014). Figure 2 details this encoding for both cases: internal node and leaf node. For the case of an internal node, the first array contains the weights' vectors of the different considered metrics in each visited node; while the second array contains the splitting rule threshold. For the case of a leaf node, the first array has the same composition as the case of an internal node but it ends with an additional cell containing a $NULL$ value to indicate that the current node is a leaf; while the second array is the same as the case of an internal node but just it ends with a boolean cell where 1 means a smelly class and 0 means the opposite. In summary, supposing that a node has an index i in the breadth-first order, its left child is located at index $2i$ and its right child is located at the $(2i+1)$ position. It is worth noting that the metrics' weights (i.e., features' weights) are defined in the interval [-10, 10] and the threshold is settled according to an efficient discretization strategy inspired by (Krętowski & Grześ, 2005), which is detailed next in subsection 2.2.1. Such setting is adopted based on the recommendations of ODT users (Bot & Langdon, 2000). Another important characteristic of our adopted encoding is that feature (metric) selection is performed in an implicit way since ignored metrics are assigned a weight of zero. In other words, the selection of metrics intervenes at the node level. Indeed, each node of the ODT is a combination of weighted metrics. Thus, if a node combines two metrics, then each of the other (non-selected) metrics are implicitly assigned a zero as weighting



(a) Smell detectors generation



(b) Detection and Identification tasks

Figure 1: The global schema of the ADIODE approach.

coefficient. According to Di Nucci et al. (2018), the software metrics (features) do not equally contribute in the detection of code smells. Moreover, as the ODT has a maximum depth (that is equal to 7 in this work according to Table 3), it is frequent to face the case that some metrics are not selected in all nodes of the ODT, and thus each of them is implicitly assigned a weight of zero in all nodes. It is worth noting that any metric could be selected in one ODT and not selected in another. This could be seen as an implicit selection of features (metrics) performed by crossover and mutation (and random initialization at the genesis step of the EA). More specifically, the EA defines in a guided stochastic way the oblique hyper-planes by selecting and weighting the adequate features in each node during the ODT induction optimization. As the ODT induction process is guided by the F-measure, the obtained detectors would be able to detect smells with imbalanced data. Similarly, for the identification case, each smell type involves a subset of metrics and not all of them. In our study, we considered the detection of eight smell types; each requiring a specific set of metrics. From a software engineering viewpoint, this implicit selection of metrics could be very useful because the detection of a particular smell (e.g., Blob) may need a set of metrics that is considerably different from the set of metrics needed for the detection of another (e.g. Data Class).

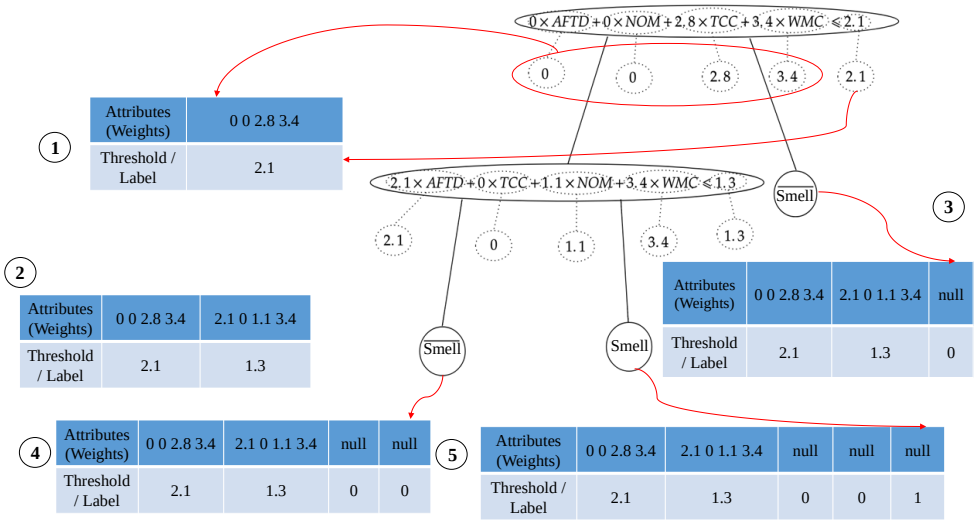


Figure 2: Oblique decision tree and its corresponding structure (Inspired by (Jankowski & Jackowski, 2014)).

2.2. GA evolution operators

The adaptation of the GA to our code smell detection problem requires to define a set of operators that are: (1) Population initialization, (2) Fitness Assignment, (3) Mating selection, (4) Crossover, and (5) Mutation. It is important to note that the reproduction operators should ensure the feasibility of the generated offspring ODTs and the fitness function should be robust to the data imbalance problem. Also, we notice that the evaluation of an ODT necessitates its execution on the BE and then its performance is computed using a 5-*fold cross-validation* strategy.

2.2.1. Population initialization and threshold generation

The GA starts by the initialization of N detectors (ODTs) as follows. The metrics' weights are within the interval [-10, 10] as previously noted in subsection 2.1.2; where ignored metrics are assigned a weight of zero. For the case of detection, labels take either 1 or 0 to refer to a smelly-class or not, respectively. For the identification case, each label contains an integer indicating the smell type. For both tasks, detection and identification, the left leaf-node (corresponding to a less-or-equal operator "$\leqslant$") is assigned randomly a label, which indicates the existence of the smell or not. Eventually, the corresponding right leaf-node is assigned the opposite label.

One of the main issues in existing smell detection approaches is the threshold definition. To the best of our knowledge (Ouni et al., 2013; Sahin et al., 2014; Mansoor et al., 2017), most existing works did not detail how to specify the threshold value and just report that it is optimized, from one iteration to the next, during the detectors generation process. Unfortunately, such strategy could generate *ineffective (meaningless)* thresholds (cf. Figure 4), which deteriorate not only the effectiveness but also the

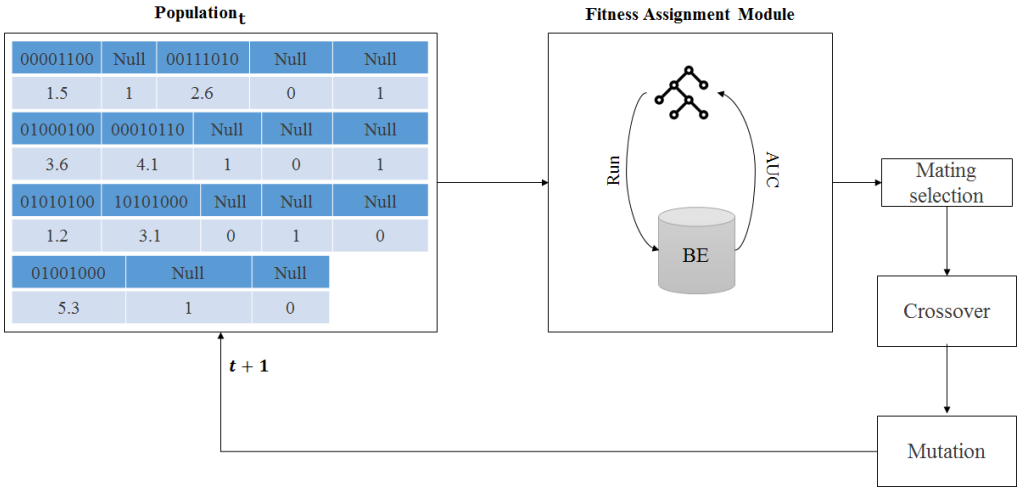


Figure 3: Main loop of the smell detectors generation module.

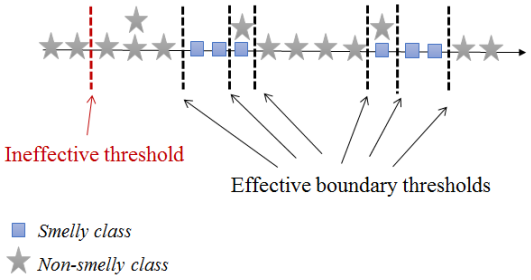


Figure 4: The identification of the boundary thresholds for a given feature.

efficiency of the tree induction process. To solve this issue, we use in ADIODE an adaptation of an existing discretization method, proposed by (Krętowski & Grześ, 2005), for efficacious threshold definition. This method has already shown interesting results when embedded within the DT induction algorithm C4.5. The working principle of this discretization technique is as follows. Once a weighted-feature combination is generated within a particular node, we consider this combination as a new (constructed) feature in the dataset and we compute all its values for every instance (software class). This computation should be done for both: smelly-classes and non-smelly ones. Afterwards, all instances are sorted in an ascending order with respect to the (constructed) feature values. Besides, boundary thresholds are identified as shown by Figure 4. Indeed, a boundary threshold for the considered feature is defined as a midpoint between such a successive pair of examples in the sequence sorted by the increasing value of this feature, so that one of the examples is *positive* and the other is *negative*. Finally, the threshold value of the considered node is randomly selected from *the set of effective generated thresholds* as depicted by Figure 4. It is worth noting that this discretization strategy is executed whenever a new feature combination is generated within a particular node. This means that the execution of this strategy could be called either at the initialization step or at the offspring generation process through crossover or mutation.

Table 1: Outcomes of two different classifiers behaving differently on the same BE.

Actual class	Predicted class			
	Classifier A		Classifier B	
	Positive	Negative	Positive	Negative
Positive (Smelly) =10	1	9	9	1
Negative (Non-smelly) = 10000	3000	7000	7000	3000

2.2.2. Fitness assignment operator

This operator assigns a quality value to each ODT of the GA's population. It is triggered after the initialization of the population and whenever a new offspring ODT is born using crossover or mutation. The quality (fitness) value will be later used in the selection process. Many researchers have used the accuracy metric (shown by Eq (1)) as a fitness function in evolutionary classification (Ma & Wang, 2009). Formally, TP (*True Positives*) is the number of actual smelly classes correctly classified, TN (*True Negatives*) is the number of actual non-smelly classes correctly classified, FN (*False Negatives*) is the number of actual non-smelly classes misclassified as smelly ones, and FP (*False Positives*) is the number of actual smelly classes misclassified as non-smelly ones.

$$Acc = \frac{TP + TN}{TP + FP + TN + FN} \tag{1}$$

$$Acc(A) = \frac{(TP = 1) + (TN = 7000)}{(TP = 1) + (FP = 3000) + (TN = 7000) + (FN = 9)} = 70\% \tag{2}$$

$$Acc(B) = \frac{(TP = 9) + (TN = 3000)}{(TP = 9) + (FP = 7000) + (TN = 3000) + (FN = 1)} = 30\% \tag{3}$$

When facing imbalanced data, the adoption of the accuracy as a fitness function is not a good choice at all for the following reason. Assuming we have two classifiers A and B (i.e., detectors), we compute the confusion matrices for each of these classifiers as represented by Table 1. We note that the exhibited example considers only three smell types (i.e., the minority instances are the smelly classes). From an accuracy viewpoint, the two classifiers A and B behave differently as their accuracy values are 70% and 30% (as shown by Eq. (2) and Eq. (3) respectively). However, from an AUC viewpoint (Eq. (4)), there is a very significant difference between the behaviors of the two classifiers. For instance, the AUC values of A and B are 0.015 and 0.315 (shown by Eq. (7) and Eq. (8) respectively). We observe that A is slightly better than B in terms of accuracy, while B is much better than A in terms of AUC. This could be explained by the fact that A has a quite good performance on the classification of the non-smelly (majority) instances and a very poor performance on the detection of smelly (minority) instances; which makes A unsuitable for smell detection. The opposite observation is seen for classifier B as it detected 9 smells out of 10.

$$AUC = \sum_{i=1}^{N-1} \frac{1}{2} \times (FPR_{i+1} - FPR_i) \times (TPR_{i+1} + TPR_i) \tag{4}$$

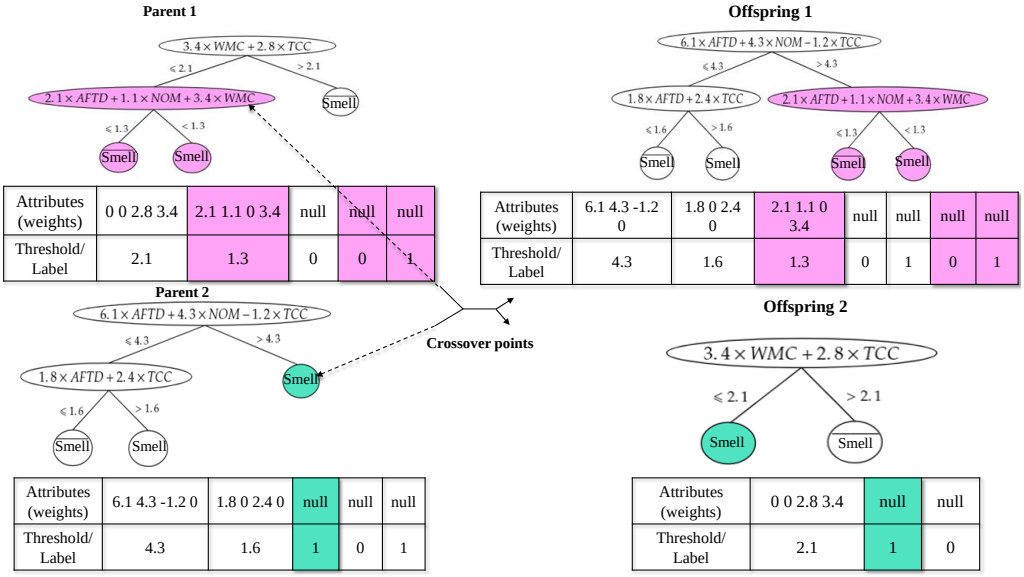


Figure 5: One-point crossover operator

$$TruePositiveRate(TPR) = \frac{TP}{TP + FN} \tag{5}$$

$$FalsePositiveRate(FPR) = \frac{FP}{FP + TN} \tag{6}$$

$$AUC(A) = 0.5 \times [(0.3 - 0) \times (0.1 + 0)] = 0.015 \tag{7}$$

$$AUC(B) = 0.5 \times [(0.7 - 0) \times (0.9 + 0)] = 0.315 \tag{8}$$

A similar observation has been detected for the opposite case where the minority instances correspond to the non-smelly classes and in this case eight smell types were considered.

2.2.3. Mating selection operator

As previously noted, one of the main advantages of our ADIODE method, is its ability to escape local optima and hence to approach the globally-optimal detectors. The main mechanism that ensures such behavior is the mating selection operator that we have adopted, i.e., the binary tournament selection operator (Brindle, 1980). The working principle of this operator is as follows. In order to select $(N/2)$ parents for reproduction where N is the population size, we execute a loop of $(N/2)$ iterations, where in each iteration two individuals (ODTs) are randomly selected (with replacement) and then only the fittest one is inserted into the mating pool. Such selection strategy allows good and bad individuals to be selected (e.g., two bad ODTs could be selected for tournament) with a preference bias towards good individuals. In this way, the GA

probabilistically accepts fitness degradations; which allows it to escape local optima and converge towards the globally-optimal ODT(s) (Barros et al., 2012).

2.2.4. Crossover and mutation operators

To crossover solutions, we adopt the one-point operator, which is illustrated by Figure 5 for our case. First, a cut-point is randomly generated with the aim to locate a subtree in each of the two parents. Then, both subtrees are exchanged. For the mutation operator, we may have two kinds of changes:

- Weight change: The operator varies the weights of the metrics by allowing the possibilities of introducing a new metric (its weight passes from zero to non-zero value) and removing an existing metrics (its weight becomes zero) (See Figure 6(a)).
- Label change: The operator switches a pair of labels, which corresponds to an exchange between two leaf-nodes (see Figure 6(b)).

Three levels of mutation are possible. The first level applies only a weight change. The second level applies only a label change. The third level applies both changes. Each level of mutation corresponds to level of degree of a randomly generated probability p such as: *Low* degree ($0 < p \leqslant 0.3$) for the first level of mutation, *Middle* degree ($0.3 < p \leqslant 0.6$) for the second level of mutation, and *High* degree ($0.6 < p \leqslant 1$) for the third level of mutation.

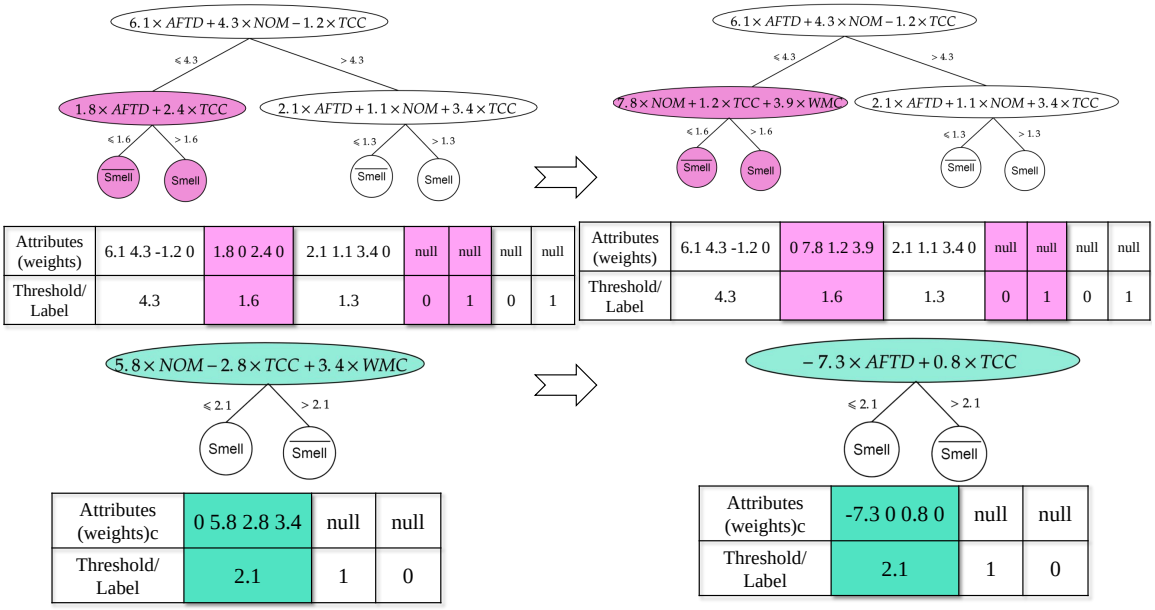
2.3. Smell detectors application module

Once the detectors are generated by the GA, the ADIODE tool is ready to be applied on unseen software systems for both cases: detection and/or identification. Figure 7 illustrates the process of labelling the N classes of a particular software system. As the generated detectors could output different labels, the majority voting strategy (Suen, 1990; Suen et al., 1992) is applied to have a robust decision, as described by Figure 7(a). To help the software engineer to identify the smell type with more precision, it is recommended to apply the identification task on the already detected smelly classes. Figure 7(b) represents such process, which uses also the majority voting strategy. As we have type-specific detectors, a smelly class could be assigned multiple labels; which means that the analyzed class has more than one smell type.

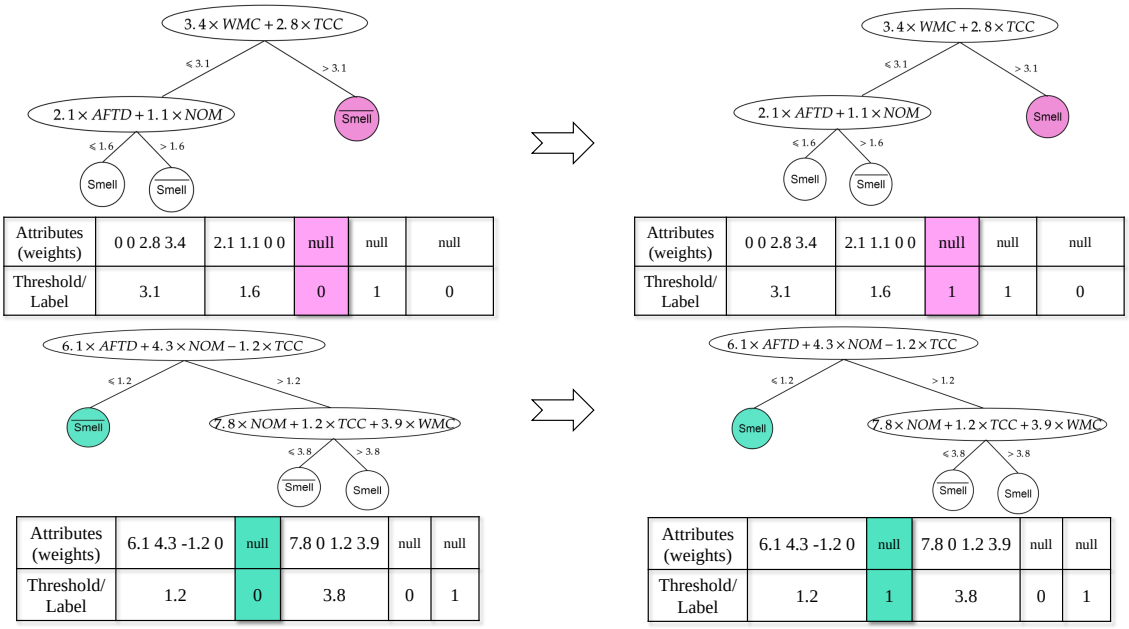
3. Experimental Validation

This section is devoted to assess the performance of our ADIODE approach with respect to the state of the art. To do so, we address the following research questions through a series of comparative experiments on six commonly-used software systems:

- **RQ0**: *How the data imbalance problem could occur in code smells detection?* To answer this question, we show how the number of smells in a particular software system increases not only with the raise of the number of considered smell types



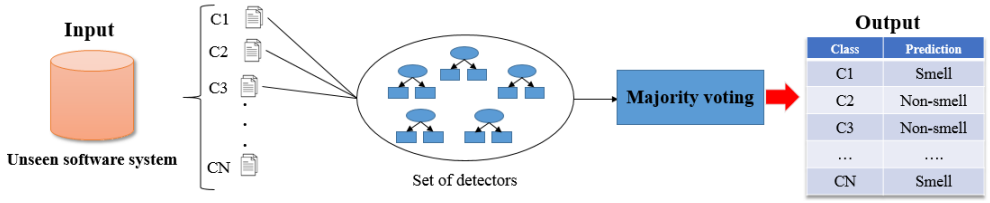
(a) Weight change



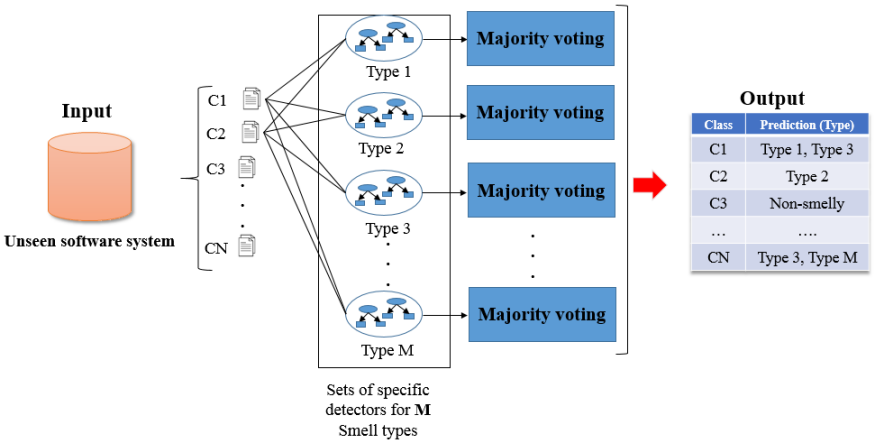
(b) Label change

Figure 6: Mutation

but also the choice of the smell types, as some smell types are more frequent then others. We study the evolution of the number of smells on six open-source software systems using 1, 2, 4, 6, and 8 smell types.



(a) Detection task



(b) Identification task

Figure 7: Illustration of the application of ADIODE on an unseen software application.

- **RQ1**: *How is the performance of* ADIODE *when facing the data imbalance problem in code smells detection?* To answer this question, we show the performance of the use of optimized oblique splitting and the guided automated definition of the threshold through a set of comparative experiments with respect to four state-of-the-art approaches. To do so, we consider the two following research sub-questions:
  - **RQ1.1**: *How does ADIODE perform when the minority class is composed of smelly instances?* We conduct a set of experiments with only three smell types that are Blob, Spaghetti Code, and Functional Decomposition.
  - **RQ1.2**: *How does ADIODE perform when the minority class is composed of non-smelly instances?* We perform a set of comparisons with eight smell types that are described next (cf. Table 10).

- **RQ2**: *How does* ADIODE *perform in smell type identification especially that such task is characterized with a higher imbalance ratio than the detection one?* It is important to study the performance of our approach when the data imbalance significantly increases with the smell type specification. In this setting, ADIODE is compared also to the four considered state-of-the-art approaches.

Table 2: Used Software in the experimentation

Systems	Releases	#Classes	KOLC	Description
GanttProject	V 1.10.2	245	41	Platform for the projects scheduling
ArgoUML	V 0.19.8	200	300	A tool for UML modeling
Xerces-J	V 2.7.0	991	240	Software for XML parsing
JFreeChart	V 1.0.9	521	170	Java Library for the generation
Ant-Apache	V 1.7.0	1,839	327	Java Library for the charts Java applications
Azureus	V 2.3.0.6	1,449	42	Peer to Peer (P2P) client program for sharing files

- **RQ3**: *Is* ADIODE *able to identify the eight studied code smells described in Table 10?* It is interesting to show the versatility of our approach with the respect to the ability of covering different types of code smells, with the aim to generalize the application of ADIODE with other smells types that are not considered in this research work.

3.1. Context of the Empirical Validation

We conduct a set of experiments on commonly used open-source Java software systems that are: ArgoUML[1], Xerces-J[2], GranttProject[3], Ant-Apache[4], Azureus[5], and JFreechart[6]. Table 2 reports the characteristics of the used systems, namely the release number, the description, and the size in terms of the number of classes (#classes) in addition to the number of lines of code (KLOC: thousands of code Lines). JFreechat is a powerful Java chart library specialized in the generation of professional quality charts. GanttProjects is a platform devoted to scheduling projects. ArgoUML tool is a popular open-source project for UML modeling. Ant-Apache is a Java library and is a build tool specifically designed for Java applications. Xerces-J is an open-source system, which is dedicated to XML files parsing. Azureus is an end-to-end (a.k.a., peer-to-peer) program for sharing files between users. We have chosen these systems as they are rich in terms of the number of existing code smells and they are frequently used in the empirical studies of smell detection (Fontana et al., 2016b; Palomba et al., 2018b; Azeem et al., 2019).

3.2. Parameter configuration for ADIODE

An important aspect that is often neglected in metaheuristic search algorithms is the adjustment of the algorithm parameters. It is important to know that the parameter setting remarkably affects the performance of an algorithm on a specific problem. For this reason, the default parameters of ADIODE employed in the simulations part (described in Table 3) are fixed using the *trial-and-error* technique, which is is a common practice

[1]http://argouml.tigris.org/
[2]http://xerces.apache.org/xerces-j/
[3]https://sourceforge.net/projects/ganttproject/files/OldFiles/
[4]http://ant.apache.org
[5]http://vuze.com/
[6]http://www.jfree.org/jfreechart/

Table 3: Parameter settings of ADIODE and its competitor algorithms.

Parameters	ADIODE	GP	MOGP	BLOP
Crossover rate	0.9	0.9	0.8	0.8
Mutation rate	0.1	0.5	0.20	0.05
Population size	200	100	100	30
Tree depth	7	7	7	7

so far in the evolutionary computation and SBSE fields (Karafotias et al., 2015; Boussaïd et al., 2017; Ramirez et al., 2018). Furthermore, the stopping criterion is set to 250,000 fitness evaluation for all the approaches (including ours) in order to ensure fairness of comparisons.

3.3. Selection of the baseline approaches

With the aim of comparing ADIODE with state of the art techniques, we selected four baselines: these are GP (Ouni et al., 2013), MOGP (Mansoor et al., 2017), BLOP (Sahin et al., 2014), and DECOR (Moha et al., 2010). The selection of these baseline approaches was based on two main reasons. On the one hand, there are three categories of search-based methods: (1) Mono-objective methods, (2) Multi-objective ones, and (3) Bi-level ones; GP, MOGP, and BLOP are three effective representative methods from these categories, respectively, and allow us to have a wider overview of how our technique works in comparison to existing approaches. On the other hand, DECOR is chosen to be the representative of the heuristic rule-based category; we included such as baseline because of our willingness to understand whether ADIODE is actually able to outperform a basic detection approach which relies on heuristics computed on the basis of the values of source code metrics. In what follow, we give a short description of the working principle of each baseline method:

1 - GP: This method evolves a set of IF-THEN rules using the genetic programming metaheuristic that communicates with a BE of code smells. Each solution is represented as a tree of detection rules; where internal nodes contains the metrics and their corresponding thresholds, while leaf nodes indicates the class labels. These rules are evolved by maximizing the number of detected defects in comparison to the expected ones in the BE. According to the reported precision and recall values by the authors, GP achieved an average F-measure of 88% on six software systems, while considering only three smell types. In the same paper, the authors proposed a multi-objective refactoring method based on NSGA-II to remove as possible the detected smells.

2 - MOGP: This method is selected as a representative of the multi-objective approaches. From a solution representation viewpoint, the MOGP uses the same encoding as GP. The main difference with GP is related to the fitness function. In fact, in MOGP NSGA-II is used to evolve the trees by optimizing two conflicting

objectives: (1) Maximizing the detection of code smells of the BE and (2) Minimizing the detection of well-designed code fragments. To achieve this goal, NSGA-II communicates with two BEs. The first one contains smells and the second one contains well-designed codes. The authors justified the use of the good codes by the fact that the use of smells alone does not allow the coverage of all smells. Thus, by maximizing the distance to a good code example, a code fragment could be considered as a suspicious anti-pattern. According to the reported precision and recall results, the average F-measure is 87 % on seven software systems while considering five smell types.

3 - BLOP: This method was proposed to solve the issue of the lack of diversity that could have a BE. To do so, the authors proposed to model the code smell detection as a bi-level optimization problem as follows. The upper level evolves a set of detection of rules, while the lower level generates a set of artificial code smells. From a fitness viewpoint, the upper level maximizes the detection of real and artificial code smells, while the lower level minimizes the likelihood of the event that the artificial smells would be detected by the upper level rules. In this way, there is a competition between both levels that aims to: (1) generate rules with important detection ability and (2) produce unseen artificial smells that diversify the BE. Based on the reported precision and recall values, the average F-measure is 90 % on nine systems while considering seven smells.

4 - DECOR: This method is a heuristic-approach rather than a search-based solution. More specifically, DECOR uses a set of rules, called "rule card"[7], that describe the intrinsic characteristics of a class affected by a smell. As an example, a Blob class is detected when a class has an LCOM5 (Lack of Cohesion Of Methods) (Henderson-Sellers, 1995) higher than 20, a number of methods and attributes higher than 20, a name that contains a suffix in the set { *Process*, *Control*, *Command*, *Manage*, *Drive*, *System* }, and it has a one-to-many association with data classes. The authors of this technique showed that DECOR can identify smells with an average F-measure of ≈80% (Moha et al., 2010).

3.4. Performance metrics

As we are solving an imbalanced data classification problem, the used performance metrics should be able to evaluate a detection method in such setting. To achieve this goal, we have chosen the F-measure (Van Rijsbergen, 1979) and the AUC metrics. The F-measure is given by equation (9) and corresponds to harmonic mean of $Precision$ (Eq. 10) and $Recall$ (Eq. 11). This choice is justified by the fact that for the case where the minority instances are the smelly classes a high F-measure value means high precision and recall values on the minority data. In this way, a high F-measure value means that a high number of detected classes are really smelly and a high number of real smells are covered by the rules. The same justification is applicable for the case where

[7]http://www.ptidej.net/research/designsmells/

the minority instances are the non-smelly classes. In fact, in case where the non-smelly classes are the minority ones, a high F-measure value means that a high number of detection are really non-smelly classes and a high number of real non-smelly ones are detected. As the F-measure is a threshold-dependent measure (Azeem et al., 2019), we complement our experimental analysis by a threshold-free metric that is the AUC (cf. equation (4)), which depicts how well a classifier approximates the trade-off between the minority (rare) instances and the majority (prevalent) ones over various classification thresholds. The AUC is well-suited for the class imbalance problem since it is a function of the true positive rate ad the false positive one, where the positive class is the minority class (Palomba et al., 2018a,b; Fontana et al., 2016b).

$$F - measure = 2 \times \frac{(Precision \times Recall)}{(Precision + Recall)} \tag{9}$$

$$Precision = \frac{TP}{TP + FP} \tag{10}$$

$$Recall = \frac{TP}{TP + FN} \tag{11}$$

3.5. Adopted statistical testing methodology

Since GAs are stochastic optimizers, they usually output different results on the same software system (or problem) from one run to another. In such settings, the comparison between stochastic smell detection approaches becomes difficult as the winning approach may vary from one run to another. To cope with the stochastic nature of results, researchers have proposed the use of statistical tests to detect any significance between the obtained results (Arcuri & Briand, 2014). Two kinds of tests are possible: (1) Parametric tests which require data normality and (2) Non-parametric tests. To avoid the issue of data normality, we have chosen to use the *Wilcoxon* test (Conover & Conover, 1980) in a pairwise fashion, with the goal to reject the null hypothesis H_0; which means that both median values of both compared algorithms over the number of runs are not significantly different. We use a significance rate α equals to 5%, which means that there is only a probability of 0.05 for rejecting H_0 while it is true. In addition to significance, it is important to report the effect size to quantify the difference between algorithms' results. Unfortunately, the *Wilcoxon* test allows verifying whether the results are statistically different or not. However, it does not give any idea about the difference magnitude. The effect size could be computed by using the Cohen's d statistic (Cohen, 1988). The effect size is considered: (1) small if $0.2 \leqslant d < 0.5$; (2) medium if $0.5 \leqslant d < 0.8$, or (3) large if $d > 0.8$.

3.6. Analysis of the Results

This subsection is devoted to report and explain the obtained comparative results with the aim to answer the three above research questions and show the effects of the main characteristics of our ADIODE approach that are: (1) the use of oblique splitting hyper-planes, (2) the ability to escape local optima, (3) the good structuring of smells

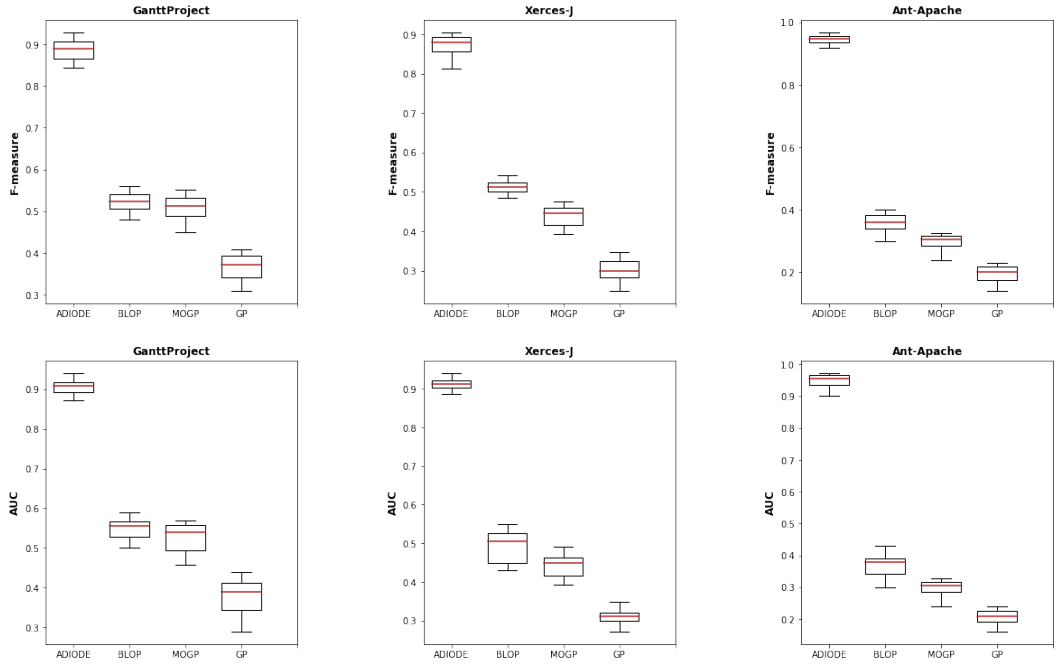


Figure 8: Box plots of F-measure and AUC values for the detection of three code smells on three different systems: *GanttProject* (small size), *Xerces-J* (medium size), and *Ant-Apache* (large size).

detectors, and (4) the informed process of threshold definition. Moreover, we show how ADIODE could be used for both tasks: detection and identification.

3.6.1. Results for RQ0

The detection of code smells mainly depends on the number of considered smell types. Table 4 presents the distribution of the number of smells per type in the six considered software systems while considering only three smell types that are among the most frequently encountered and studied (Palomba et al., 2018b). These smell types are Blob, Spaghetti Code, and Functional Decomposition. We observe from this table that the minority instances are the smelly classes, while the majority instances are non-smelly classes. According to the data imbalance ratio values given in the last column, we see that the data distribution is significantly imbalanced. This could be explained by the fact that the number of non-smelly classes largely exceeds the number of smelly ones. For small-size systems (GattProject and ArgoUML), the ratio varies between 19.80 % and 21.00 %. For medium-size systems (JFreeChart and Xercess-J) and large-size systems (Ant-Apache and Azureus) the ratio respectively belongs to [16.69 %, 17.25 %] and [09.35 %, 10.79 %]. It is worth noting that the lower the ratio is, the higher the data distribution imbalance is, since the ratio could be seen as the percentage of minority instances with respect to all instances. Based on the imbalance ratio analysis, we could say that the imbalance is higher when the system size is larger. This could be explained by the fact that the introduction of new classes to an existing software system version increases the probability of introducing new anti-patterns; which makes the patterns

Table 4: Results of the detection task in the case of three smell types (Blob, Spaghetti Code (SC), and Functional Decomposition (FD)).

	Blob	SC	FD	Classes	Smelly classes	Non-smelly classes	Imbalance ratio with 3 smell types (Blob, SC, FD)
GranttProject	11	16	17	245	44	201	17.95 %
ArgoUML	30	0	0	200	30	170	15.00 %
Xercess-J	100	23	16	991	139	852	14.02 %
JFreeChart	40	18	10	521	68	453	13.05 %
Ant-Apache	130	13	0	1839	143	1696	07.77 %
Azureus	100	21	19	1449	140	1309	09.66 %
Overall	411	91	62	5245	564	4681	10.75 %

Table 5: Results of the detection task in the case of eight smell types (Blob, Data Class (DC), Feature Envy (FE), Long Method (LM), Duplicate Code (DuC), Long Parameter List (LPL), Spaghetti Code (SC), and Functional Decomposition (FD)).

	Blob	DC	FE	LM	DuC	LPL	SC	FD	Classes	Smelly classes	Non-smelly classes	Imbalance ratio with 8 smell types
GranttProject	11	35	8	29	50	32	16	17	245	198	47	19.18 %
ArgoUML	30	20	15	53	0	40	0	0	200	158	42	21.00 %
Xercess-J	100	201	105	90	190	95	23	16	991	820	171	17.25 %
JFreeChart	40	100	72	54	115	25	18	10	521	434	87	16.69 %
Ant-Apache	130	240	317	247	360	360	13	0	1839	1667	172	09.35 %
Azureus	100	300	260	120	210	200	81	19	1449	1290	159	10.97 %
Overall	411	896	777	593	925	752	151	62	5245	4567	678	12.92 %

more complex for software engineers.

Table 5 exhibits the same statistics as Table 4 but for a different case consisting in considering eight smell types. According to the statistics values, the situation is now the opposite: The minority instances are non-smelly classes and the majority ones are the smelly classes. The imbalance ratio, consisting in this case as the number of non-smelly classes divided by the total number of classes, lies within the interval [09.35 %, 21.00 %]. Again, the larger the system is, the more probable the imbalance increases. The incorporation of new classes within a particular system version increases the probability of new smells occurrences; thereby making the number of non-smelly classes lower, which sharpen the data imbalance.

Both Table 4 and Table 5 illustrate two cases of smells detection. However, if we consider the smell type identification problem, the imbalance would be much higher. Let us consider the case of Blob, which is one of the most frequent and studied smell types in both academy and industry. The imbalance ratio when considering only Blob lies within [04.48 %, 06.90 %]. Moreover, if consider all software systems simultaneously, the number of Blob instances is 411 and the total number of classes is 5245. This shows that data imbalance is higher in identification then in detection. We can imagine the case of less frequent smells such as Functional Decomposition. In summary, the data imbalance problem is a frequently encountered issue in code smells detection and needs special algorithmic methods to be handled.

3.6.2. Results for RQ1.

This section reports the results for our first research question. For the sake of comprehensibility, we discuss them by considering each sub-research question independently.

Results for RQ1.1. To answer RQ1.1, we conduct a set of experiments on the six considered software systems while considering only three smell types (Blob, Spaghetti Code, and Functional Decomposition), which define an imbalanced environment where the minority class contains the smelly instances. The rise of the imbalance ratio may in turn cause the increase of the probability of occurrence of small disjuncts. Figure 9 illustrates the small disjuncts that correspond to underrepresented sub-groups within a particular class (Holte et al., 1989). These subgroups are usually difficult to classify especially when using orthogonal splitting hyper-planes. The results are analyzed in terms of the above-described metrics that are the F-measure and the AUC. Table 6 presents the values of these two metrics for the five compared methods in addition to the statistical significance and effect sizes. Based on this table, our ADIODE method outperforms all the three considered peer search-based approaches in addition to DECOR. For instance, ADIODE F-measure values span between [89.21, 93.80] while the second-best method (BLOP) values are between 37.6 and 56.81. The MOGP outputted results are similar to BLOP ones. The GP did not perform well in such an imbalanced environment as its F-measure values lie within [20.61, 38.74]. The AUC values are conform to the F-measure results in terms of significance and magnitude.

The above results confirm the outperformance of ADIODE over the three considered search-based methods. These results could be explained as follows. First, the used fitness functions of BLOP, MOGP, and GP are not well-suited for the case of imbalanced data, which may mislead the detection rules' search process. Second, the threshold definition in these three methods is randomly generated, which may conduct to ineffective splitting values. Indeed, when the threshold is randomly defined, it is highly probable that such a threshold gives rise to an empty sub-class or to a sub-class containing all the instances of the parent node. Finally, these three methods implicitly use axis-parallel splitting hyper-planes, which is not efficient in an imbalanced classification environment.

The DECOR method generated the worst results. Indeed, its F-measure belong to [10.21, 23.58] and its AUC varies within [0.11, 0.24]. Such poor results could be explained by the static a priori definition of the detection rules, which is not effective at all in the case of imbalanced smell data. In fact, in an imbalanced environment, the manual metrics' thresholds definition is a very fastidious and complex task.

In a nutshell, the obtained results of the F-measure and AUC measures for the four peer considered methods in our experimentation could be explained by the fact that the TPR (the ratio of correctly detected smelly classes) is low while the FPR (the ratio of incorrectly detected smelly classes: false alarms) is high. Such results prove that these methods are unable to correctly detect the smelly classes. This could be explained by the fact that the positive data class is the smelly one and consequently such systems could output false alarms to the developers by recommending smelly classes as non-smelly ones, which may mislead the software engineer. Moreover, as the number of non-smelly classes is relatively high, a high FPR could have a very negative impact on

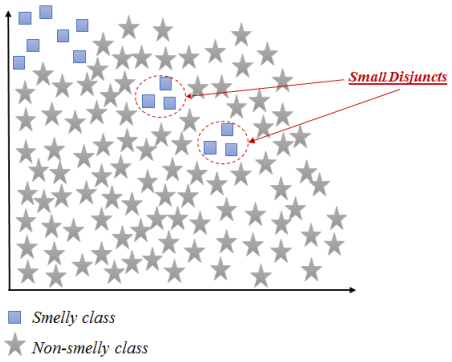


Figure 9: Example of small disjuncts problem in the case of imbalanced data.

the software developer task by suggesting non-smelly classes as smelly ones. Such false suggestions could mislead the developer and push him/her to introduce new smells in some non-smelly classes.

To have an idea about the difference magnitude, we report the effect sizes in Table 6. According to this table, ADIODE has, on average, medium effect sizes on small-size systems (Gantt Project and ArgoUML) and large effect sizes on the other four systems over the three search-based approaches (BLOP, MOGP, GP). This can be explained by the fact that ADIODE is able to find good detectors for imbalanced data, while it is not the case for BLOP, MOGP, and GP especially when the system size increases. Indeed, the increase in the number of system's classes and the consideration of a few smell types usually make the imbalance ratio higher, which makes finding good detectors harder.

All the experimental results related to RQ1.1 are confirmed by the box plots illustrated by Figure 8 on one small-size, one medium size, and one large-size systems. The observation of these box plots shows the outperformance of ADIODE with respect to the three search-based methods in terms of significance and effect sizes. Actually, the box plots of ADIODE are significantly above the three others with a considerable difference, which confirms the statistical significance and the magnitude. These three methods use axis-parallel splitting hyper-planes, which is not efficient in an imbalanced classification environment.

Results for RQ1.2. To answer RQ1.2, we conduct a set of experiments on the six open source software systems with taking into account eight smell types described in Table 10, which define an imbalanced environment where the minority instances are non-smelly. We recall that in this case the data imbalance problem is due to the high-number of smelly classes with respect to non-smelly ones. As previously noted, the increase of the imbalance data ratio may in turns entail the rise of the probability of occurrence of small disjuncts. Based on Table 7, our ADIODE method outperforms all the three considered peer search-based approaches in addition to DECOR. For instance, ADIODE F-measure values span between [91.23, 95.24] while the second-best method (BLOP) values are between 37.6 and 55.80. The MOGP results are similar to BLOP ones. The GP did not perform well in such an imbalanced data environment as its

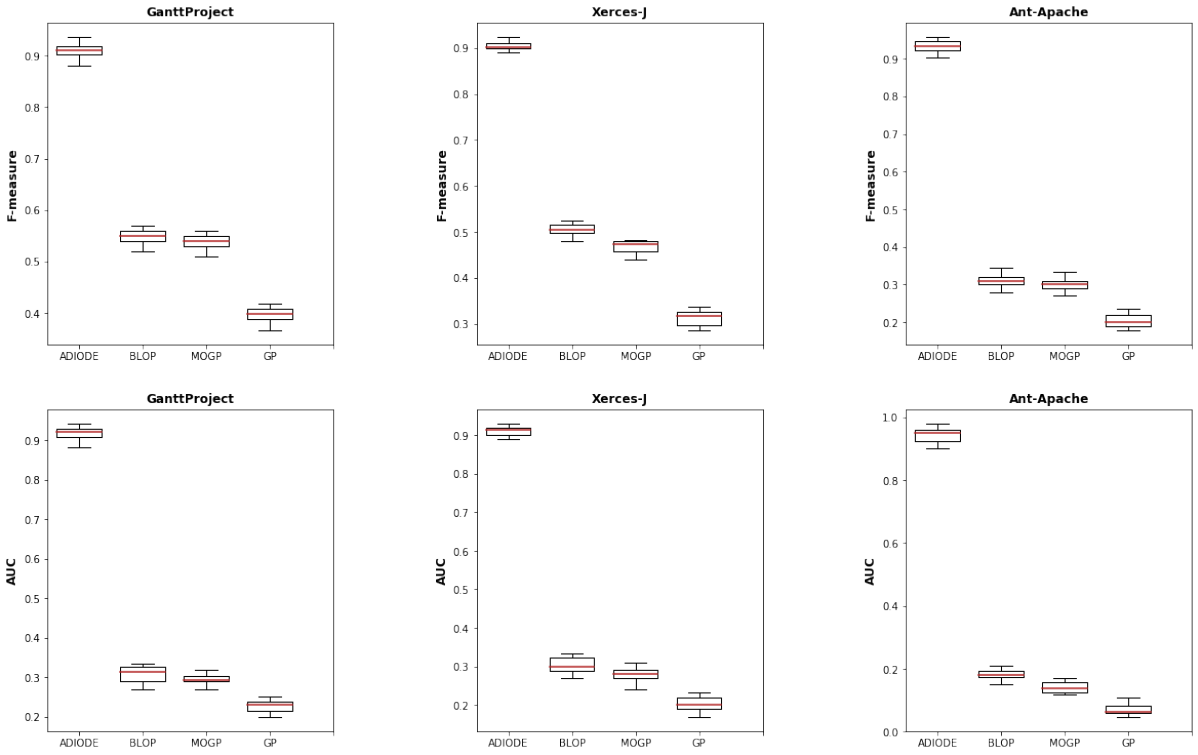


Figure 10: Box plots of F-measure and AUC values for the detection of eight code smell types on three different systems: *GanttProject* (small size), *Xerces-J* (medium size), and *Ant-Apache* (large size).

F-measure values lie within [22.53, 40.56]. DECOR generated the worst results. Indeed, its F-measure belong to [11.20, 35.34] and its AUC varies within [0.12, 0.04].

The moderate degradation of the F-measure and the extreme degradation of the AUC for the four peer considered methods could be explained by the fact that the TPR (the ratio of correctly detected non-smelly classes) and the FPR (the ratio of incorrectly detected non-smelly classes) are poor. Such poor values significantly degrade the F-measure results and the AUC values as confirmed by Table 7. Although these peer methods may recommend a high number of smelly classes to the software developer, such recommendations are just a hazardous result since the majority instances are the smelly ones.

All the experimental results related to RQ1.2 are confirmed by the box plots of Figure 11 on three systems with different sizes. The analysis of these box plots demonstrates the superiority of ADIODE over the three search-based methods in terms of significance and effect sizes. Actually, the box plots of ADIODE are above the three others with a considerable difference, which confirms the statistical significance and magnitude.

3.6.3. Results for RQ2.

The aim of this subsection is to asses the performance of the compared methods on the smell type identification problem. Such a problem is more difficult than the detection

Table 6: F-measure and AUC median scores of ADIODE, DECOR, GP, MOGP, and BLOP over 31 runs of the detection task for three smell types (Blob, Spaghetti Code, and Functional Decomposition). The sign "+" at the i^{th} position signifies that the algorithm F-measure (or AUC) median value is statically different from the i^{th} algorithm value. The sign "-" means the opposite. Similarly, the effect sizes values (small (s), medium (m), and large (l)) using the Cohen'd statistics are given. Best F-measure (or AUC) values are in Bold. Second-best F-measure (or AUC) values are underlined.

Systems	ADIODE		DECOR [a]		GP		MOGP		BLOP	
	F-measure %	AUC	F-measure %	AUC	F-measure %	AUC	F-measure %	AUC	F-measure %	AUC
Gantt Project	**89.21** (++++) (l l l l)	**0.908** (++++) (l l l l)	23.58 (+++) (m l l)	0.240 (+++) (m l l)	38.74 (++) (m m)	0.394 (++) (m m)	53.7 (-) (s)	0.544 (-) (s)	<u>54.3</u>	<u>0.558</u>
ArgoUML	**90.43** (++++) (l l l l)	**0.9078** (++++) (l l l l)	34.33 (-++) (s l l)	0.351 (-++) (s l l)	35.85 (++) (l l)	0.369 (++) (l l)	55.95 (-) (s)	0.568 (-) (s)	<u>56.81</u>	<u>0.576</u>
Xercess	**89.3** (++++) (l l l l)	**0.917** (++++) (l l l l)	17.48 (+++) (l l l)	0.164 (+++) (l l l)	30.19 (++) (m l)	0.3136 (++) (m l)	45.26 (+) (m)	0.468 (+) (m)	<u>50.75</u>	<u>0.5102</u>
JFreeChart	**91.5** (++++) (l l l l)	**0.916** (++++) (l l l l)	15.49 (+++) (l l l)	0.1528 (+++) (l l l)	26.43 (++) (m l)	0.2704 (++) (m l)	38.17 (+) (m)	0.3926 (+) (m)	<u>48.23</u>	<u>0.4937</u>
Azureus	**92.25** (++++) (l l l l)	**0.957** (++++) (l l l l)	10.79 (+++) (l l l)	0.1192 (+++) (l l l)	22.71 (++) (m l)	0.236 (++) (m l)	31.57 (+) (m)	0.321 (+) (m)	<u>40.91</u>	<u>0.41</u>
ApacheAnt	**93.8** (++++) (l l l l)	**0.961** (++++) (l l l l)	10.21 (+++) (l l l)	0.110 (+++) (l l l)	20.61 (++) (m l)	0.219 (++) (m l)	30.5 (+) (m)	0.313 (+) (m)	<u>37.6</u>	<u>0.3887</u>

[a] The rules' thresholds of DECOR are manually defined

Table 7: F-measure and AUC median scores of ADIODE, DECOR, GP, MOGP, and BLOP over 31 runs of the detection task for the eight smell types. The sign "+" at the i^{th} position signifies that the algorithm F-measure (or AUC) median value is statically different from the i^{th} algorithm value. The sign "-" means the opposite. Similarly, the effect sizes values (small (s), medium (m), and large (l)) using the Cohen'd statistics are given. Best F-measure (or AUC) values are in Bold. Second-best F-measure (or AUC) values are underlined.

Systems	ADIODE		DECOR [a]		GP		MOGP		BLOP	
	F-measure %	AUC	F-measure %	AUC	F-measure %	AUC	F-measure %	AUC	F-measure %	AUC
Gantt Project	**91.43** (++++) (l l l l)	**0.9276** (++++) (l l l l)	35.34 (-++) (s m m)	0.12 (+++) (m m m)	40.56 (++) (m m)	0.2415 (-+) (s m)	55.42 (-) (s)	0.30 (-) (s)	<u>55.80</u>	<u>0.328</u>
ArgoUML	**92.87** (++++) (l l l l)	**0.9302** (++++) (l l l l)	35.86 (-++) (s m m)	0.1705 (–+) (s s m)	36.04 (++) (m m)	0.1877 (-+) (s m)	55.28 (-) (s)	0.24 (-) (s)	<u>57.78</u>	<u>0.3273</u>
Xercess	**91.23** (++++) (l l l l)	**0.9273** (++++) (l l l l)	19.71 (+++) (m m l)	0.1010 (+++) (m m l)	32.04 (++) (m m)	0.2085 (–) (s s)	47.09 (-) (s)	0.2879 (-) (s)	<u>51.42</u>	<u>0.3171</u>
JFreeChart	**95.24** (++++) (l l l l)	**0.9573** (++++) (l l l l)	17.08 (-++) (s m l)	0.0972 (-++) (s m l)	30.63 (++) (m m)	0.1271 (++) (m m)	40.06 (-) (s)	0.2095 (-) (s)	<u>50.03</u>	<u>0.2264</u>
Azureus	**94.01** (+++-) (l l l l)	**0.9431** (++++) (l l l l)	12.97 (+++) (m m m)	0.0820 (-++) (s m m)	24.15 (++) (m m)	0.1096 (-+) (s m)	35.60 (-) (s)	0.1853 (-) (s)	<u>42.33</u>	<u>0.2170</u>
ApacheAnt	**94.73** (++++) (l l l l)	**0.9516** (++++) (l l l l)	11.20 (+++) (m m m)	0.0486 (-++) (s m m)	22.53 (++) (m m)	0.0893 (-+) (s m)	32 (-) (s)	0.1456 (-) (s)	<u>32.10</u>	<u>0.1960</u>

[a] We used the original version of DECOR, which works only on three smell types: Blob, SC, and FD.

one because the imbalance ratio value of the identification task is higher than the one of the detection task. To realize the comparative experiments, for each type smell, we merge all software systems' classes into the same BE and then define the smelly classes that composes the minority class. During this task, we have observed that the number of non-smelly classes largely exceeds the number of smelly ones, which entails a higher imbalance in the BE.

Table 8 reports the F-measure and AUC values of ADIODE, BLOP, MOGP, GP, and DECOR. We observe from this table that our ADIODE method considerably outperforms the other methods in terms of both performance measures. For ADIODE, the F-measure values lie between [86.26, 94.50] and the AUC values belongs to [0.86, 0.95]; while the second-best algorithm, BLOP, has an F-measure value between 20.74 and 43.22 and an AUC value between 0.20 and 0.44. These results could be explained as follows. When the imbalance ratio increases, the probability of occurrence of small disjuncts increases too. Hence, for the identification task, we may have a minority class that is composed of different clusters. To find such minority clusters that could be geographically dispersed over the feature space, where features correspond to the software metrics given in Table 11, the splitting hyper-planes and the thresholds should be very effective.

As we have adopted the ODT as a structure for our detectors, each node in ADIODE detectors is a weighted combination of features. Such combinations define oblique partitioning splits in the feature space that could be suited to any geometrical shape of the minority class including the possible disjuncts that may appear. In contrast, BLOP, MOGP, and GP use only orthogonal splits; which makes finding the smelly class instances more difficult for them. In addition to splitting hyper-planes, the high data imbalance problem necessitates to quickly finding meaningful and effective splitting thresholds. This could not be achieved by the peer three search-based approaches as their thresholds are randomly produced. As previously noted, such random generation could engender meaningless splits as we may obtain empty data sub-classes or ineffective splits (that do not separate data subsets) and this could be simulated through Figure 4. ADIODE does not face such problems since its thresholds are defined using the informed process described by Figure 4, which makes all its splits effective. It important to note that again the fitness functions of BLOP, MOGP, and GP are not well-fitted to imbalanced data and this issue becomes more serious and critical with the raise of the imbalance ratio.

Figure 11 displays the box plots of the compared algorithms for three types of smells that are Blob, Feature Envy, and Functional Decomposition. According to this figure, we see that ADIODE succeeds to achieve a high performance in terms of F-measure and AUC on the three different anti-patterns. This figure clearly shows the versatility of our method with regard to the smell type. The performance of the other methods is relatively poor and this could be explained by the high imbalance ratio when considering the detection of a single smell type (i.e., the identification problem). The box plots are compliant with the effect sizes reported in Table 8. For Blob and Data Class smells, the ADIODE effect sizes are usually large and sometimes medium. This could be explained by the important frequency of such kind of smells. For the six other smells,

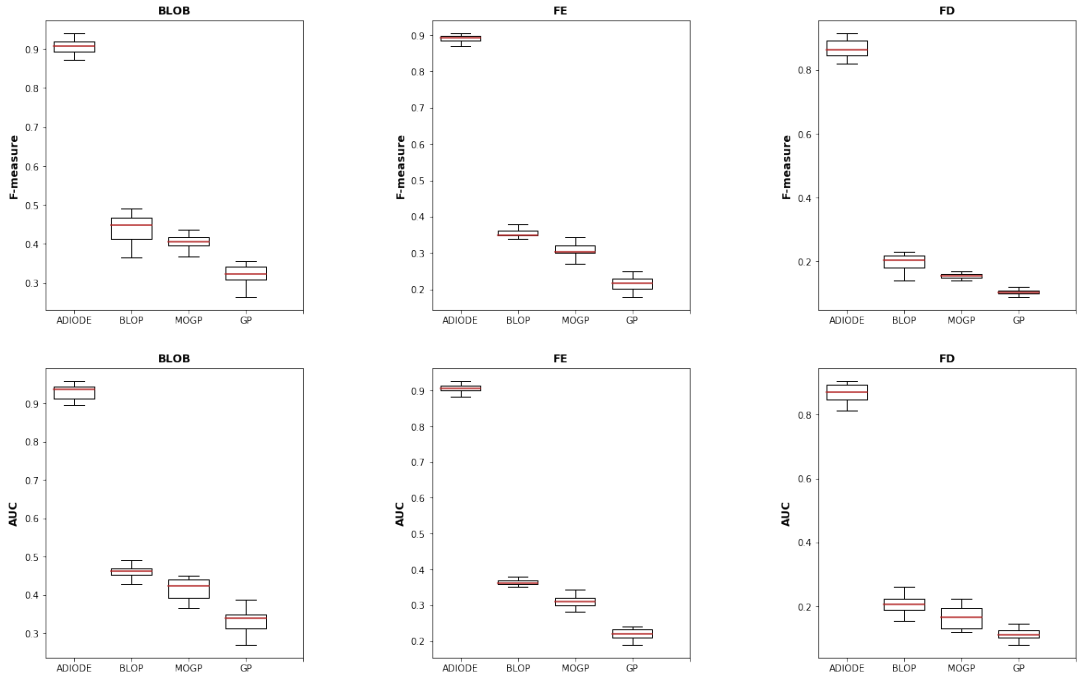


Figure 11: Box plots of F-measure and AUC values for the identification of three code smell types (Blob, Feature Envy (FE), and Functional Decomposition (FD)).

the ADIODE effect sizes are usually large especially for the case of Spaghetti codes and Function decomposition, since these two smell types' occurrence frequencies are low. It is important to note that DECOR obtained the poorest results in the identification comparisons and this could be due to the use of manually predefined rules.

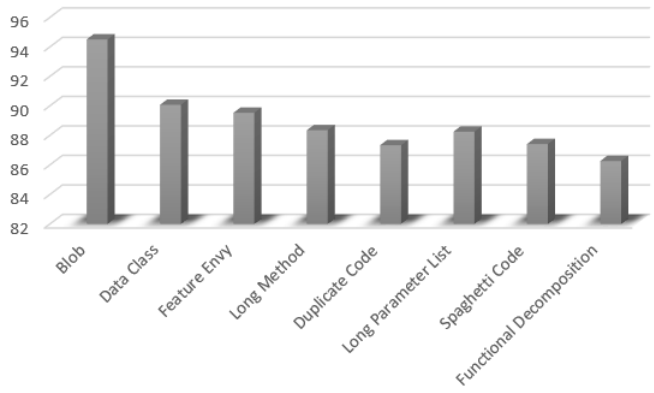
3.6.4. Results for RQ3

The goal of this subsection is to demonstrate the versatility of ADIODE on the detection of different kinds of smells. Figure 12 plots two histograms showing the F-measure and AUC variations over the eight considered smell types in this study. From Figure 12(a), we observe that the distribution of the F-measure values varies between 86% and 94% without presenting a high variation over the different smell types. The same observation could be seen for the AUC from the histogram of Figure 12(b). The AUC values are between 0.86 and 0.95 and their variance is enough small over the smell types. We can conclude from these two histograms that our ADIODE method is a generic tool that could be used with different types of smells.

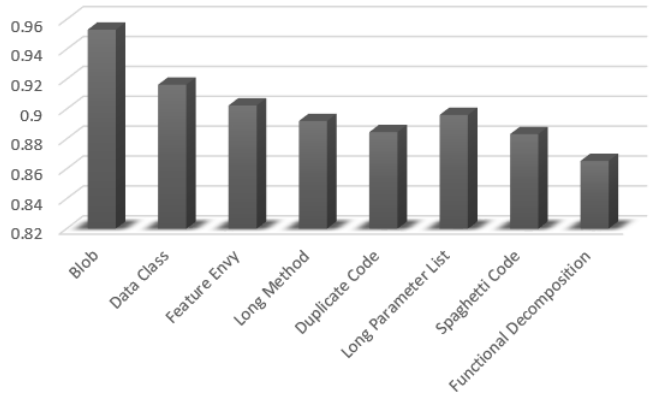
According to the metrics' values of both histograms, we could cluster the eight smell types into three clusters: Blob, Data Class, Feature Envy, Long Method, Duplicate Code, Long Parameter List, and Spaghetti Code, Functional Decomposition. These clusters are ranged according to the number of occurrences of the smell types in the BE. Eventually, if other anti-patterns are considered in ADIODE, this distribution may change. Nevertheless, even if this change occurs, our ADIODE tool could be

Table 8: F-measure and AUC median scores of ADIODE, DECOR, GP, MOGP, and BLOP over 31 runs of the identification task. The sign "+" at the i^{th} position signifies that the algorithm F-measure (or AUC) median value is statically different from the i^{th} algorithm value. The sign "-" means the opposite. Similarly, the effect sizes values (small (s), medium (m), and large (l)) using the Cohen'd statistics are given. The N/A means that the given approach is Not Applicable (N/A) on the corresponding smell. Best F-measure (or AUC) values are in Bold. Second-best F-measure (or AUC) values are underlined.

Code Smell	ADIODE		DECOR		GP		MOGP		BLOP	
	F-measure %	AUC	F-measure %	AUC	F-measure %	AUC	F-measure %	AUC	F-measure %	AUC
Blob	**94.5** (++++) (l l l l)	**0.9531** (++++) (l l l l)	34.6 (-++) (s m m)	0.3527 (-++) (s m m)	33.16 (++) (m m)	0.3422 (++) (m m)	41.77 (-) (s)	0.4203 (-) (s)	<u>43.22</u>	<u>0.4412</u>
Data Class	**90.07** (+++) (l l l)	**0.9163** (+++) (l l l)	N/A	N/A	26.7 (++) (m m)	0.2683 (++) (m m)	32.49 (-) (s)	0.3308 (-) (s)	<u>37.13</u>	<u>0.3829</u>
Feature Envy	**89.53** (+++) (l l l)	**0.9024** (+++) (l l l)	N/A	N/A	25.16 (++) (m m)	0.2647 (++) (m m)	31.18 (-) (s)	0.3239 (-) (s)	<u>36.75</u>	<u>0.3702</u>
Long Method	**88.35** (+++) (l l l)	**0.8920** (+++) (l l l)	N/A	N/A	23.92 (++) (m m)	0.2462 (++) (m m)	30.53 (-) (s)	0.3110 (-) (s)	<u>34</u>	<u>0.3475</u>
Duplicate Code	**87.33** (+++) (l l l)	**0.8846** (+++) (l l l)	N/A	N/A	19.64 (++) (m l)	0.2030 (++) (m l)	27.59 (+) (m)	0.2801 (+) (m)	<u>32.70</u>	<u>0.3314</u>
Long Parameter List	**88.25** (+++) (l l l)	**0.8961** (+++) (l l l)	N/A	N/A	14.2 (++) (l l)	0.1454 (++) (l l)	23.16 (+) (m)	0.2414 (+) (m)	<u>29.92</u>	<u>0.2984</u>
Spaghetti Code	**87.41** (++++) (l l l l)	**0.8832** (++++) (l l l l)	08.13 (+++) (l l l)	0.0802 (+++) (l l l)	11.30 (++) (m l)	0.1214 (++) (m l)	19 (+) (m)	0.1984 (+) (m)	<u>25.36</u>	<u>0.2605</u>
Functional Decomposition	**86.26** (++++) (l l l l)	**0.8653** (++++) (l l l l)	06.35 (+++) (l l l)	0.064 (+++) (l l l)	10.22 (++) (m l)	0.1110 (++) (m l)	15.49 (+) (m)	0.1608 (+) (m)	<u>20.74</u>	<u>0.2021</u>



(a) F-measure histogram plot



(b) AUC histogram plot

Figure 12: Median F-measure and AUC scores for the eight code smell types in the identification process.

applied in a generic manner as it bases the generation of detectors on the BE. The main characteristics of ADIODE, which are the effective oblique hyper-planes, the efficient threshold definition, and the imbalance-oriented fitness function (AUC), are preserved whatever are the smell types present in the BE.

4. Discussion and Implications

The results of our empirical study provided a number of insights and practical implications for the research community that need further discussion.

- **The problem of code smell detection is highly imbalanced.** The findings coming from **RQ0** clearly point out the high imbalance between classes affected and not by code smells. The problem seems to be independent from the specific type of code smell, i.e., we identified imbalance for both detection and identification

tasks, and the imbalance ratio depends on the number of considered smell types. In the first place, this result confirms what Pecorelli et al. (2019b) discovered in the context of machine learning-based code smell detection mechanisms. Secondly, our findings lead to a clear implication for the research community: researchers working in the field of code smell detection have to deal with data imbalance. Indeed, to create effective solutions, they are required to design intelligent solutions that take the distribution of code smells into account in their algorithmic behaviors. The technique proposed in this paper represents a first attempt toward explicitly considering data imbalance in the code smell detection/identification process, which indeed leads to better performance with respect to the baselines. Nonetheless, we believe that further approaches should be devised and, perhaps more importantly, previously devised techniques might be revisited in order to include an algorithmic component that could deal with the imbalance information. This would potentially provide developers with an empowered set of detection/identification techniques that would allow them to better perform quality assurance of their projects.

- **Toward adaptive mechanisms for data balancing.** When analyzing the results achieved for **RQ0**, we noticed that the imbalance problem becomes more evident as the size of a system increases and, indeed, in large systems the imbalance almost doubles the one of small systems. Furthermore, this problem is even more evident when considering the individual code smell types. While confirming again the need for considering the data imbalance problem in code smell detection/identification, these findings may possibly lead researchers to devise brand new balancing strategies to deal with the problem. As a matter of fact, the number of code smells affecting a software system may be notably lower than other kinds of problems like, for instance, software defects - which have been estimated by previous work in affecting up to 70% of source code classes (Catolino et al., 2018; Di Nucci et al., 2017; Pascarella et al., 2019). This aspect directly challenges the fundamentals of data balancing theories, making the problem of code smells peculiar and hard to treat. At the same time, it also creates interesting future research avenues aimed at not only devising specific data balancing approaches for code smell detection/identification, but also possible adaptive methods that allow to deal with data imbalance based on the relative size of the codebase analyzed - this goes along the lines of having dedicated instruments to deal with software engineering problems, which is at the basis of the rising research area of software engineering for artificial intelligence (Amershi et al., 2019; Zhang & Tsai, 2003).

- **On the value of Oblique Decision Trees.** The key idea behind the proposed technique concerns with the adoption of Oblique Decision Trees, which were previously shown as effective to deal with data imbalance (Murthy et al., 1994; Wickramarachchi et al., 2016). In our context, we mixed together machine learning and evolutionary algorithms in order to evolve a set of ODTs that will later form the instruments enabling the actual detection/identification of code smells. The peculiarities of ODTs made them particularly suitable for our context—as shown

in **RQ1**-**RQ3**—confirming previous findings reported in literature. Additionally, we were able to substantially contribute to the field of code smell identification, which still represents one of the main challenges in research (Fontana et al., 2016a; Palomba et al., 2018a). From a practical perspective, our findings lead to two main implications. First, we provided a new methodology to detect code smells, which represents one the first combining machine learning and search-based algorithms: researchers can therefore build upon it to keep improving the way code smells can be detected and identified or even to measure its ability in dealing with different code smell types. Perhaps more importantly, we developed a *technique that engineers can implement within their projects to empower the quality assurance processes in place.*

5. Threats to Validity

This section discusses the factors that influence our empirical study. These factors can be divided into three categories: internal, external, and construct validity. Threats to internal validity concern the correctness of the experimentations' results of our proposal, while threats to external validity are related to the generalizability of the generated results. Finally, threats to construct validity are concerned with the relationship between the theory and the observation.

- *Threat to internal validity:* Over this work, the main internal threat is related to the parameter setting of our EA since the latter has a stochastic behavior, meaning that it outputs different results from one run to another even if we use the same parameters' values. This makes the parameter setting a challenging task. In our work, we have used the commonly adopted method in the literature that is the trial-and-error method and this is done for all algorithms under comparisons. It is worth noting that parameter tuning is still so far a major challenge in the metaheuristics community and the SBSE one (Huang et al., 2020), because we cannot predict the performance of a stochastic algorithm before its execution; as opposed to exact (deterministic) algorithms.

- *Threats to external validity:* In this work, the experiments are based on six open-source Java projects with different sizes and domains (cf. Table 2). However, we cannot affirm that our ADIODE tool would preserve its performance for the cases of other programming languages and other software technologies (e.g., Web services, mobile applications, etc.). As such, further replications of our work in other contexts might be worthwhile.

- *Threats to construct validity:* In our experimental study, the BE is built using three tools that are PMD[8] (Gopalan, 2012), DECOR (Moha et al., 2010), and JDeodorant (Tsantalis & Chatzigeorgiou, 2009). As the definition (expression) of any quality metric could change from one tool to another, the metric definition

[8]https://pmd.github.io/

could be seen as a source of a construct validity threat. Preliminary experiments have been conducted in Appendix F to show the versatility of ADIODE with respect to the metrics' definitions' changes. This has been done by constructing a new BE using the inFusion tool [9] and then assessing the performance of ADIODE on this new BE. The preliminary results are promising because modifying the metrics' definitions has a direct impact on the content of the BE and not ADIODE. However, more detailed experiments with other bases of examples built with diversified tools and procedures are needed to further study the possible effects of metrics' definitions' changes. It is important to note that, in industrial contexts, the BE is usually constructed using not only rule-based tools such as DECOR and inFusion, but also subjective opinions of human experts that basically rely on their background knowledge and experiences. This could be the source of another threat consisting in the disagreement between experts' opinions about the smelliness of a particular class or the definition of a particular smell.

6. Related Work

The detection of code smells has been dealt with in several studies from various perspectives that shows the employment of several detection techniques. The techniques to detect smells could be broadly grouped into four main categories. These are *Rules/heuristic-based approaches*, *Search-based approaches*, *Machine Learning-based approaches*, and *Others*. Notwithstanding, there is almost no attempt on the identification of code smells in case of imbalanced data. Table 9 summarizes the characteristics and the hyper-parameters of existing smell detection methods, with the aim to show the main distinguishing features that makes ADIODE able to deal with imbalanced data with regard to existing works. Based on this table, the following observations could be concluded:

- Only two works from the literature have considered the data imbalance issue that are Fontana et al. (2016b) and Di Nucci et al. (2018). Unfortunately, these works did not propose specific algorithmic strategies to handle imbalance, since they only used stratified random sampling to rebalance data. Thus, these two works focused on the data-level by means of data preprocessing and not on the algorithm-level. It is important to note that data rebalance through under-sampling or over-sampling is not easy at all and should be performed with a high level of prudence to avoid misleading the induction process of the classifier (Das et al., 2018). One the one hand, under-sampling removes some instances from the majority class, which could incur a considerable loss of information. On the other hand, over-sampling replicates minority instances or generates new artificial ones to increase the size of the minority class, which may cause redundancy, information divulgation, and/or noise.

[9] Available at http://www.intooitus.com/products/infusion

- ADIODE is the only method that is able to deal with data imbalance in an algorithmic manner (without using data preprocessing). This is done through the evolutionary optimization of a population of ODTs based on the F-measure that is used as a fitness function. Thus, ODTs are evaluated using a quality function that is insensitive to the data imbalance in binary classification (He & Garcia, 2009; Luque et al., 2019; Mullick et al., 2020).

- Traditional methods using the DT as a classifier employ the gain ratio as a feature selection criterion at a particular node. As this choice is greedy and could lead to locally-optimal splits, some researchers used the random forest where a set of DTs are constructed using a random selection of the feature at a particular node; then, the results over all trees are aggregated. This strategy may also generate only locally-optimal splits (and trees) because of the random generation process. However, in ADIODE, at each node of a particular ODT, a feature combination is selected using the *crossover* and *mutation* operators (Neither the gain ratio nor the random selection is employed. Only the random selection is executed once at the initialization of the population for the first generation). On the one hand, the crossover operator exploits the fittest parts of the parents with the aim to inherit the good feature combinations, which could be seen as a type of constructed features (Hammami et al., 2019). On the other hand, the mutation operator allows diversifying the feature combinations of the ODTs in order to not get stuck in locally-optimal choices. We conclude that ADIODE performs a wise feature selection at DT nodes by means of the process of guided-randomness that is based on three characteristics: (1) the probabilistic acceptance of worse choices to escape local optima, (2) the inheritance of fit building blocks from parent trees for the construction of offspring trees, and (3) the convergence process towards the globally-optimal DT that is guided by the F-measure. In summary, we can say that the node feature selection is performed by crossover and mutation.

- The column TDM shows the efficient threshold generation performed by ADIODE thanks to the use of Kretowski-&-Grzes method. In fact, the entropy-based discretization is greedy and hence leads to locally-optimal thresholds. The use of the mutation operator for this task is not a wise choice because the probability of the generation of meaningless and/or ineffective splitting thresholds is high. The manual definition is still subjective and depending on the expert knowledge and experience. The choice of the Kretowski-&-Grzes method is motivated by two reasons. On the one hand, as illustrated by figure 4, the execution of this method at each node of the ODT generates a set of effective thresholds for the related constructed feature (expressing the oblique split of the feature combination). On the other hand, the choice of the split is not performed using a greedy selection based on the gain ratio (such as the case of C4.5), but the threshold is randomly selected from the set of effective splitting thresholds produced by the Kretowski-&-Grzes method, thereby allowing escaping locally-optimal thresholds. By default, this threshold generation mechanism is performed in an automated way using the

Kretowski-&-Grzes method. However, if the software engineer would like to interact with ADIODE (e.g., after a user-defined number of generations), the set of effective thresholds for any node of any ODT could be exposed to him/her, and then he/she could choose or calibrate the threshold based on not only the exposed thresholds generated by the Kretowski-&-Grzes method but also his/her personal experience. We recall that there are no specific circumstances to choose between: (1) generated threshold selection or (2) threshold calibration. The software engineer has the freedom to make a choice among these two options. Once the interaction is up, the ADIODE method continues the ODTs optimization process until either meeting the stopping criterion or triggering a new interaction.

- Existing SBSE approaches model the detector as an ad-hoc tree of rules and do not report the tree depth. This is an important factor because the detectors will be used later by humans and thus a high depth makes the understandability of the detection rule tree very difficult. In ADIODE, the depth of the ODT is set to seven. This value is settled after several experiments to get a good compromise between detection rate and human comprehensibility of the detector.

- Detection methods based on rules use two types of pruning: (1) pre-pruning where the nodes are pruned during the induction and (2) post-pruning where nodes are pruned after the whole tree generation. SBSE methods, including ADIODE, implicitly execute pruning by means of crossover. This operator could stop the development of branches at any node during the evolution process through sub-tree exchange. It is important to note that a node is a sub-tree that is composed of a single root node without children.

- The last column of the table shows that, although many fitness functions have been proposed to optimize detection rules, all of them are sensitive to data imbalance. The use of the F-measure as a fitness function in ADIODE is another distinguishing attribute of our tool thanks to the ability of this classification quality measure to deal with imbalanced data, as mentioned from the beginning of this paper.

6.1. Rules/heuristic-based approaches

The initial attempts to identify the software classes contaminated by anti-patterns have concentrated on the definition of *rule/ heuristic-based* approaches, which depend upon metrics to catch detours from good *OO* design practices. First, Erni & Lewerentz (1996) employed metrics to assess the performance of frameworks with the aim of enhancing them. The authors used the concept of multi-metrics where the m-tuple of metrics represent a quality criterion. Later, Marinescu (2004) suggested a detection strategy that relies on metric-based method to analyze code source and to detect the defects in the *OO* design fragments at different levels such as method, class, and subsystem. In this context,Lanza & Marinescu (2007) express the detection strategy as a combination of a set of metrics with thresholds to define a set of rules for 11 anti-patterns. These rules are a set of metric-threshold pairs connected by logical operators(AND/OR). However, these mentioned heuristics have been implemented in a tool called *InCode* (Marinescu et al.,

Table 9: Comparison between code smells detection methods based on their characteristics and hyper-parameters (DIC: Data Imbalance Consideration, CM: Classifier Model, ASI: Adaptation Strategy to Imbalance, NFSC: Node Feature Selection Criterion, TDM: Threshold Definition Method, MaxTD: Maximum Tree Depth, PS: Pruning Strategy, CEFF: Classifier Evaluation Fitness Function, N/R: Not Reported, N/A: Not Applicable).

Category	Smell detection methods	DIC		CM	ASI	NFSC	TDM	MaxTD	PS	CEFF
		No	Yes							
Rules/heuristic based approaches	Marinescu (2004)	x		Manual designed heuristic rule	Nothing	N/A	N/A	N/A	N/A	N/A
	Lanza & Marinescu (2007)	x		Manual designed heuristic rule	Nothing	N/A	N/A	N/A	N/A	N/A
	Marinescu et al. (2010)	x		Manual designed heuristic rule	Nothing	N/A	N/A	N/A	N/A	N/A
	Moha et al. (2010)	x		Manual designed heuristic rule	Nothing	N/A	N/A	N/A	N/A	N/A
	Tsantalis & Chatzigeorgiou (2011) and Fokaefs et al. (2012)	x		Manual designed heuristic rule	Nothing	N/A	N/A	N/A	N/A	N/A
Machine Learning based approaches	Kreimer (2005)	x		Decision tree (C4.5)	Nothing	Gain Ratio	Entropy based discretization	N/R	Pre-pruning using Gain Ratio	N/A
	Amorim et al. (2015)	x		Decision tree (C5.0)	Nothing	Gain Ratio	Entropy based discretization	N/R	Pre-pruning using Gain Ratio	N/A
	Khomh et al. (2009)	x		Belief Bayesian Network	Nothing	N/A	N/A	N/A	N/A	N/A
	Khomh et al. (2011)	x		Belief Bayesian Network	Nothing	N/A	N/A	N/A	N/A	N/A
	Vaucher et al. (2009)	x		Belief Bayesian Network	Nothing	N/A	N/A	N/A	N/A	N/A
	Maiga et al. (2012a,b)	x		Support Vector Machine	Nothing	N/A	N/A	N/A	N/A	N/A
	Hassaine et al. (2010)	x		Artificial Immune System based classifier	Nothing	N/A	N/A	N/A	N/A	N/A
	Oliveto et al. (2010)	x		B-Splines based classifier	Nothing	N/A	N/A	N/A	N/A	N/A
	Fontana et al. (2016b)		x	Pruned Decision tree (C4.5: Two versions: with and without Boosting)	Data stratified random sampling	Gain Ratio	Entropy based discretization	N/R	Pre-pruning using Gain Ratio	N/A
				Unpruned Decision tree (C4.5: Two versions: with and without Boosting)					Nothing	
				Post-pruned Decision tree with reduced error (C4.5: Two versions: with and without Boosting)					Post-pruning using Error Reduction	
				JRip classifier (RIPPER: Two versions: with and without Boosting)		N/A	N/A	N/A	Post-pruning using Error Reduction	
				Random Forest (Random construction: Two versions: with and without Boosting)		Random feature selection	N/A	N/R	Nothing	
				Naïve Bayes (Two versions: with and without Boosting)		N/A	N/A	N/A	N/A	
				SVM (SMO (RBF and Polynomial): Two versions: with and without Boosting)		N/A	N/A	N/A	N/A	
				4 SVM variances (Linear, Polynomial, Radial, and Sigmoid) (Two versions: with and without Boosting)		N/A	N/A	N/A	N/A	
	Di Nucci et al. (2018)		x	The same set of classifiers as Fontana et al. (2016b)	Normalized data stratified random sampling					

Category	Smell detection methods	DIC		CM	ASI	NFSC	TDM	MaxTD	PS	CEFF
		No	Yes							
Search-based approaches	Kessentini et al. (2011)	x		Ad-hoc Rule Tree	Nothing	Crossover-Mutation	Mutation	N/R	Crossover	NB_{TP} [a]
	Ouni et al. (2013)	x		Ad-hoc Rule Tree	Nothing	Crossover-Mutation	Mutation	N/R	Crossover	F_{norm} [b]
	Boussaa et al. (2013)	x		Ad-hoc Rule Tree	Nothing	Crossover-Mutation	Mutation	N/R	Crossover	$F_{coverage}$ and $Cost$ [c]
	Kessentini et al. (2014)	x		Ad-hoc Rule Tree	Nothing	Crossover-Mutation	Mutation	N/R	Crossover	F_{GP} and F_{GA} [d]
	Sahin et al. (2014)	x		Ad-hoc Rule Tree	Nothing	Crossover-Mutation	Mutation	N/R	Crossover	F_{upper} and F_{lower} [e]
	Mansoor et al. (2017)	x		Ad-hoc Rule Tree	Nothing	Crossover-Mutation	Mutation	N/R	Crossover	F_1 and F_2 [f]
	Our proposed ADIODE		x	Oblique Decision Tree	Oblique Decision Tree evolution using F-measure	Crossover-Mutation	Krętowski & Grześ Method	7	Crossover	F-measure
Others	Rapu et al. (2004)	x		Manual designed heuristic rule	Nothing	N/A	Manual designed heuristic choice	N/R	N/A	N/A
	Palomba et al. (2013, 2015)	x		Manual designed heuristic rule	Nothing	N/A	Manual designed heuristic choice	N/R	N/A	N/A
	Fu & Shen (2015)	x		Manual designed heuristic rule	Nothing	N/A	Manual designed heuristic choice	N/R	N/A	N/A
	Emden & Moonen (2002)	x		Manual designed heuristic rule	Nothing	N/A	Manual designed heuristic choice	N/R	N/A	N/A
	Langelier et al. (2005)	x		Manual designed heuristic rule	Nothing	N/A	Manual designed heuristic choice	N/R	N/A	N/A
	Dhambri et al. (2008)	x		Manual designed heuristic rule	Nothing	N/A	Manual designed heuristic choice	N/R	N/A	N/A

[a] $NB_{TP} = \sum_{i=1}^{p} a_i$. The NB_{TP} fitness function calculates the number of detected smells with respect to those expected in the BE (Kessentini et al., 2011)

[b] $F_{norm} = \frac{\frac{\sum_{i=1}^{p} a_i}{t} + \frac{\sum_{i=1}^{p} a_i}{p}}{2}$. The F_{norm} computizes the number of detected smells with respect to those expected in the BE. Then, the obtained value is normalized (Ouni et al., 2013).

[c] $F_{coverage} = r + \frac{\frac{\sum_{i=1}^{p} a_i}{t} + \frac{\sum_{i=1}^{p} a_i}{p}}{2}$ and $Cost = Max(\sum_{j=1}^{w} \sum_{k=1}^{l} |M_k(d_i) - M_k(C_j)|) + z$. The first fitness function $F_{coverage}$ (for the first population) calculates the number of detected defects comparing to those expected in the BE as well as the produced "Artificial" defects by the second level. However, $Cost$ refers to the cost function that asses the quality of a generated solution (a set of generated defects). (Boussaa et al., 2013).

[d] $F_{GP} = \frac{f_{coverage}(S_i) + f_{intersection}(S_i)}{2}$ and $F_{GA} = \frac{cost_(O_j) + f_{intersection}(O_j)}{2}$. The F_{GP} calculates the number of detected defects in comparison to the expected ones in the BE and also with those detected by the GA. In contrast, the F_{GA} fitness function evaluates the resulted detectors using the dissimilarity score among detectors and various reference code fragments (with the aim to assess the diversity) (Kessentini et al., 2014).

[e] $F_{upper} = \frac{\frac{Precision(S,BE) + Recall(S,BE)}{2} + \frac{\#detectedArtificialCodeSmells}{\#artificialCodeSmells}}{2}$ and $F_{lower} = t + Min(\sum_{j=1}^{w} \sum_{k=1}^{l} |M_k((cArtificialCS) - M_k(cReferenceCode)|)$. The F_{upper} evaluates the coverage of defect examples and the coverage of the produced "artificial" defect at the lower level. The F_{lower} calculates the number of non-detected artificial defects that are produced based on the distance with the well-designed examples (Sahin et al., 2014).

[f] $F_1 = \frac{\frac{|DCS(x)| \cap |ECS|}{|ECS|} + \frac{|DCS(x)| \cap |ECS|}{|DCS(x)|}}{2}$ and $F_2 = \frac{\frac{|DCS(x)| \cap |EGE|}{|EGE|} + \frac{|DCS(x)| \cap |EGE|}{|DCS(x)|}}{2}$. The F_1 maximizes the coverage of the detect of code smell examples, while F_2 minimizes the detection of well-designed examples (Mansoor et al., 2017).

2010). In another work, Moha et al. (2010) proposed the DECOR approach that contains the main steps for the specification and the detection of code smells. This approach begins by describing the symptoms of the defects with the employment of an abstract rule language. These descriptions imply various concepts (e.g. class roles and structures), which are mapped to the detection algorithm. Recently, other approaches adopted the clustering methods to detect code smells. According to this consideration, Tsantalis & Chatzigeorgiou (2009) suggested a tool called *JDeodorant* that could identify the smells and recommend a set of move method refactoring operations. At the beginning the tool is tuned to detect the *Feature Envy* bad smells, then it is able to support other bad smells (e.g., *Blob*, *Long Method*, and *State checking*) (Tsantalis & Chatzigeorgiou, 2011; Fokaefs et al., 2012). The detection mechanism in *JDeodorant* tool relies on the code metrics, which are then linked together according to the supervised clustering methods and thresholds to cut-off the obtained dendrograms.

6.2. Machine Learning-based approaches

Recently, a new trend is concerned with the use of machine learning methods for the detection of code smells. The methods in this part are almost supervised category ones. More specifically, these methods are built using a training data and then they are performed on software projects to predict the smelliness of a class. Kreimer (2005) initially proposed an adaptive detection model that combines known methods to find the occurrences of design flaws viz., God Class and Long Method based on code metrics as features to train Decision Trees (DTs). The evaluation were performed on two small software systems: IYC and WEKA. After 10 years, Amorim et al. (2015) confirmed the preceding findings by assessing the performance of DTs on four different software systems for the detection of 12 anti-patterns.

Khomh et al. (2009) proposed the use of Bayesian Networks to identify the occurrences of the Blob anti-patterns on open source projects (GanttProject and Xerces-J). Later, Khomh et al. (2011) extended their work to a novel approach named BDTEX (Bayesian Detection Expert) that is validated on God Class, Functional Decomposition, and Spaghetti Code. The proposed approach relied on the Goal Question Metric to build the Belief Bayesian Networks (BBNs) in order to derive information from the code smells' definition. The detection outputs of BDTEX are likelihoods that are assigned to the smelly code components rather than boolean values. Besides, Vaucher et al. (2009)

relied on the BBNs to study the lifecycle of the Blob' evolution and thus, differentiating between truly God Classes from those happened by accident (i.e., bad code).

Maiga et al. (2012a,b) presented an SVMDetect approach that detects anti-patterns using Support Vector Machine (SVM). This approach has been validated with four anti-patterns (Blob, Functional Decomposition, Spaghetti Code and Swiss Army Knife) on three open-source software systems: ArgoUML, Azureus, and Xerces. Afterward, the authors extended their previous approach to another more accurate one called SMURF that considers the participants' feedbacks.

Hassaine et al. (2010) applied the immune-inspired approach to detect the Blob smells. The proposed approach is developed to systematically detect the smells in classes, which violates the characteristics of some created rules. Similarly, Oliveto et al. (2010) proposed an approach called ABS (Anti-pattern identification using B-Splins) that relied on the numerical analysis technique to detect the smelly instances.

In recent works, the authors studied the performances of various ML methods for the detection of code smells. Fontana et al. (2016b) conducted an investigation of the performances of 16 supervised ML methods accompanied by their boosting variant for the detection of four different anti-patterns (Blob, Data Class, Feature Envy, and Long Method) on 74 Java software systems. Additionally, the authors applied the under-sampling technique through the training and evaluation phases in order to escape the poor performances produced by the ML techniques in case of imbalanced datasets. In other paper, Fontana & Zanoni (2017) performed a classification of code smell severity based on a multinomial classification and regression technique. However, such approach can aid the developer for prioritizing or ranking the classes or methods. In contrast, Di Nucci et al. (2018) mentioned the limitations of Fontana et al. (2016b) on the way of constructing the dataset. Based on these limitations, the authors configured Fontana' datasets and generated new ones that are well-suited for the real case scenarios.

6.3. Search-based approaches

The search-based approaches are applied in the software engineering issues to resolve several optimization problems using meta-heuristics such as *GA*, GP, and so on. The detection phase is considered as the most important phase since it reports to the next phase (refactoring) the existing code smells in the software projects before starting on the correction. For instance, the begining was with Kessentini et al. (2011), they developed an automated approach for detecting code smells in software systems based on detection rules. These rules are defined as combination of metrics along with thresholds that were extracted from comparing with various heuristic search algorithm dedicated to the extraction of rules (Harmony Search, Simulated Annealing, and Particle Swarm Optimization). The experimentation was conducted on three different anti-patterns (Blob, Spaghetti Code, and Functional Decomposition) based on two open source software systems: GanttProjeject, Xerces-J. Later, Ouni et al. (2013) introduced a search-based approach for code smell detection in OO software systems. Such approach was the first one that derivates the detection rules from the defect examples using Genetic Programming. The authors applied their approach on three anti-patterns (Blob, Spaghetti Code,

and Functional Decomposition) based on six open source projects with different sizes viz., GanttProject, Xerces-J, ArgoUML, Quick UML, LOG4J, and AZUREUS.

Boussaa et al. (2013) proposed to employ the competitive co-evolutionary search aiming to handle the problem of code smell detection. The proposed approach includes two competing populations that evolve simultaneously; the first population consists on generating a set of detection rules that maximizes the detection ratio of code smell examples, while the second one maximizes the generation of artificial created smells that are not detected by the first population. The experimental study was performed on three code smell type (Blob, Spaghetti Code, and Functional Decomposition) based on four software projects with various sizes viz., ArgoUML, Xerces, Ant-Apache, and Azureus.

Kessentini et al. (2014) suggested to introduce the parallel way to its approach called PEA (Parallel evolutionary Algorithm) for code smell detection. Over the proposed approach, the researches combined the Genetic Programming and Genetic Algorithm in a parallel way throughout the optimization step to generate a set of detection rules from examples of various code smells and also the detectors from the non-smelly (well-designed) examples of code source respectively. The PEA was tested on nine code smell types (Blob, functional decomposition, spaghetti code, feature envy, data class,long parameter list, lazy class and shotgun surgery) based on nine open source projects: JFreechart, Ganttproject, ApacheAnt V1.5.2,ApacheAnt V1.7.0, Nutch, Log4J, Lucene, Xerces-J andRhino.

Sahin et al. (2014) proposed BLOP (Bi-Level Optimization Problem) approach that relies on the bi-level optimization problem for the production of code smell detection rules throughout two levels. The first level task, named the upper-level (or the leader), is responsible for the creation of a set of detection rules in the aim to maximize the coverage of both code smell examples and the artifical code smells produced by the second level. The lower level or the follower is responsible to divulge the most possible smells that are not detected by the generated detection rules from the upper- level. The authors applied their approach to detect variety of code smells (such as Blob, Feature Envy, Data Class, Spaghetti Code, Functional Decomposition, Lazy Class, Long Parameter List) on nine Java software projects with large and medium sizes: JFreeChart, GanttProject, ApacheAnt, Nutch, Log4J, Lucene, Xerces-J, and Rhino.

Mansoor et al. (2017) introduced the multi-objective aspect on the process of the generation of dection rules that are dedicated to the detection of code smells. The paposed approach called MOGP (Multi-Objective Genetic Programming) that is utilized to find the best combination of metrics that simultaneously increases the number of detected code smell examples and reduces the number of detected non-smelly examples. The researchers evaluated their approach on five code smells (Blob, Feature Envy, Data Class, Spaghetti Code, Functional Decomposition) based on different open source Java software projects: ArgoUML v0.26, ArgoUML v0.3, Xerces-J, Ant-Apache v1.5, Ant-Apache v1.7.0, GanttProject, Azureus.

6.4. Others

Some researchers have taken into consideration the historical information regarding the evolution of code source to detect code smells. Initially, the concept starts with Rapu et al. (2004) that derived historical information from the defected structure. The authors considered a set of history measurements, which represents the evolution of smells. The obtained results were joined with the initial detection strategies. The researchers tested their approach on the detection of two code smells (Blob and Data Class) based on three open source projects which are, two inhouse projects and Jan (3D graphics environment). In another work, Palomba et al. (2013, 2015) have developed an approach called HIST (Historical Information for Smell deTection). The HIST approach consists of detecting code smells using the projects' historical information derived from the revision control system. The researchers used in the experimentation of their approach, five different code smells (Blob, Feature Envy, Divergent Change, Shotgun Surgery, and Parallel Inheritance) and eight open source software projects (and 20 software projects in the later study). Similarly, Fu & Shen (2015) proposed an approach (based on associated rule mining) that is able to derive information history (related to a modification or addition, either classes ormethods or packages) from projects where their duration of growth history is huge. In this work, the authors attempt to validate their approach by detecting three code smells (duplicated code, shotgun surgery, and divergent change) on five software projects viz., Eclipse, jUnit, Guava, Closure Compiler, and Maven.

Some authors such as Emden & Moonen (2002) focused on the visualization of code smells for complex software analysis as the jCOSMO approach. Such approach consists on parsing the java source code and displaying the defected code fragments by smells and their connection using the graphical view. Later, Langelier et al. (2005) presented a framework named VERSO in which the visualization is based on colors to represent properties and mainly to support quality analysis in software. Still, on the same features of the previous approach, Dhambri et al. (2008) proposed an approach to detect smells by automatically detecting a part of symptoms and leaving the rest to the judgment to the human analyst. The authors applied their approach on three types of design anomalies (Blob, Functional Decomposition, and Divergent Change) based on two software systems: PMD and Xerces-J.

7. Conclusion and future works

In this paper, we have proposed ADIODE as a new tool for the detection and identification of code smells. Through this research work, we have tackled a usually neglected issue in the *SBSE* community; which corresponds to the fact that smells detection/identification is an imbalanced binary classification problem. This is justified by the fact that the number of smelly classes is largely greater than the number of non-smelly ones in software systems when considering a low number of smell types, vice versa. Our approach ADIODE, consisting in an EA that evolves a set of ODTs based on a base of smell examples, have shown its merits thanks to three main features. First, the use of oblique splitting hyper-planes is more effective and efficient than the use of orthogonal ones for the case of imbalanced data, especially when small disjuncts occur.

Second, the adopted data-driven discretization strategy avoids the case of having an empty sub-class or a sub-class containing all instances. Finally, the *AUC* is chosen as fitness function thanks to its ability to faithfully evaluate detectors on both imbalanced and balanced data. The statistical analysis of the obtained results has shown the merits of ADIODE with regard to four state-of-the-art methods.

Several interesting perspectives have been detected through this work. First, we are planning to handle the problem of disparity, which means that a particular class could be assigned several smell types. Such situation introduces a problem of uncertainty and imprecision about anti-patterns types for a particular software class. Ignoring the uncertainty issue may negatively impact the detectors classification performance and consequently incurs bad decisions regarding the choice of the refactoring operation sequences that should be applied to remove as possible the detected anti-patterns. Second, another uncertainty problem in this direction is the presence of imprecision not only in target instances' labels but also within the features (i.e., metrics) of the training set. These imprecision problems are due to the absence of a clear precise rule for each smell type. Thirdly, an important issue that could occur in the smell type identification problem is the absence of target labels for a considerable part of the base of anti-pattern examples. This corresponds to a semi-supervised learning task and specific mechanisms should be designed to solve such special classification problem.

Acknowledgement

Fabio Palomba gratefully acknowledges the support of the Swiss National Science Foundation through the SNF Project No. PZ00P2_186090 (TED).

References

Al-Sahaf, H., Bi, Y., Chen, Q., Lensen, A., Mei, Y., Sun, Y., Tran, B., Xue, B., & Zhang, M. (2019). A survey on evolutionary machine learning. *Journal of the Royal Society of New Zealand*, *49*, 205–228.

Amershi, S., Begel, A., Bird, C., DeLine, R., Gall, H., Kamar, E., Nagappan, N., Nushi, B., & Zimmermann, T. (2019). Software engineering for machine learning: A case study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)* (pp. 291–300). IEEE.

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

Arcuri, A., & Briand, L. (2014). A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, *24*, 219–250.

Azeem, M. I., Palomba, F., Shi, L., & Wang, Q. (2019). Machine Learning Techniques for Code Smell Detection: A Systematic Literature Review and Meta-Analysis. *Information and Software Technology*, *108*, 115–138.

Baeza-Yates, R., Ribeiro-Neto, B. et al. (1999). *Modern information retrieval* volume 463. ACM press New York.

Barros, R. C., Basgalupp, M. P., De Carvalho, A. C., & Freitas, A. A. (2012). A survey of evolutionary algorithms for decision-tree induction. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, *42*, 291–312.

Bot, M. C., & Langdon, W. B. (2000). Application of genetic programming to induction of linear classification trees. In *Proceedings of the European Conference on Genetic Programming* (pp. 247–258). Springer.

Boussaa, M., Kessentini, W., Kessentini, M., Bechikh, S., & Chikha, S. B. (2013). Competitive Coevolutionary Code-Smells Detection. In *Proceedings of the 5th International Symposium on Search Based Software Engineering* (pp. 50–65). Springer volume 8084.

Boussaïd, I., Siarry, P., & Ahmed-Nacer, M. (2017). A survey on search-based model-driven engineering. *Automated Software Engineering*, *24*, 233–294.

Breiman, L., Friedman, J., Stone, C. J., & Olshen, R. A. (1984). *Classification and regression trees*. CRC press.

Brindle, A. (1980). *Genetic algorithms for function optimization*. Ph.D. thesis The Faculty of Graduate Studies University of Alberta.

Catolino, G., Palomba, F., De Lucia, A., Ferrucci, F., & Zaidman, A. (2018). Enhancing change prediction models using developer-related factors. *Journal of Systems and Software*, *143*, 14–28.

Catolino, G., Palomba, F., Fontana, F. A., De Lucia, A., Zaidman, A., & Ferrucci, F. (2019). Improving change prediction models with code smell-related information. *arXiv preprint arXiv:1905.10889*, .

Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on software engineering*, *20*, 476–493.

Cohen, J. (1988). *Statistical power analysis for the behavioral sciences*. Erlbaum Associates, Hillsdale.

Conover, W. J., & Conover, W. J. (1980). Practical nonparametric statistics, .

Crasso, M., Zunino, A., Moreno, L., & Campo, M. (2009). Jeetuningexpert: A software assistant for improving java enterprise edition application performance. *Expert Systems with Applications*, *36*, 11718–11729.

Cunningham, W. (1993). The WyCash portfolio management system. *ACM SIGPLAN OOPS Messenger*, *4*, 29–30.

Das, S., Datta, S., & Chaudhuri, B. B. (2018). Handling data irregularities in classification: Foundations, trends, and future challenges. *Pattern Recognition*, *81*, 674–693.

Devarriya, D., Gulati, C., Mansharamani, V., Sakalle, A., & Bhardwaj, A. (2020). Unbalanced breast cancer data classification using novel fitness functions in genetic programming. *Expert Systems with Applications*, *140*, 112866.

Dhambri, K., Sahraoui, H., & Poulin, P. (2008). Visual detection of design anomalies. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering,* (pp. 279–283). IEEE.

Di Nucci, D., Palomba, F., De Rosa, G., Bavota, G., Oliveto, R., & De Lucia, A. (2017). A developer centered bug prediction model. *IEEE Transactions on Software Engineering*, *44*, 5–24.

Di Nucci, D., Palomba, F., Tamburri, D. A., Serebrenik, A., & De Lucia, A. (2018). Detecting code smells using machine learning techniques: are we there yet? In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering* (pp. 612–621). IEEE.

Emden, E. V., & Moonen, L. (2002). Java quality assurance by detecting code smells. In *Proceedings of the 9th Working Conference on Reverse Engineering,* (pp. 97–106). IEEE.

Erni, K., & Lewerentz, C. (1996). Applying design-metrics to object-oriented frameworks. In *Proceedings of the 3rd international software metrics symposium* (pp. 64–74). IEEE.

Fokaefs, M., Tsantalis, N., Stroulia, E., & Chatzigeorgiou, A. (2012). Identification and application of extract class refactorings in object-oriented systems. *Journal of Systems and Software*, *85*, 2241–2260.

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

Fontana, F. A., Dietrich, J., Walter, B., Yamashita, A., & Zanoni, M. (2016a). Antipattern and code smell false positives: Preliminary conceptualization and classification. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)* (pp. 609–613). IEEE volume 1.

Fontana, F. A., Mäntylä, M. V., Zanoni, M., & Marino, A. (2016b). Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, *21*, 1143–

1191.

Fontana, F. A., & Zanoni, M. (2017). Code smell severity classification using machine learning techniques. *Knowledge-Based Systems*, *128*, 43–58.

Fowler, M., & Beck, K. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesely.

Fu, S., & Shen, B. (2015). Code Bad Smell Detection through Evolutionary Data Mining. In *Proceedings of the 2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement,* (pp. 1–9). IEEE.

Gopalan, R. (2012). *Automatic detection of code smells in Java source code*. Ph.D. thesis Dissertation for Honour Degree, The University of Western Australia.

Haixiang, G., Yijing, L., Shang, J., Mingyun, G., Yuanyue, H., & Bing, G. (2017). Learning from class-imbalanced data: Review of methods and applications. *Expert Systems with Applications*, *73*, 220–239.

Hammami, M., Bechikh, S., Hung, C.-C., & Said, L. B. (2019). A multi-objective hybrid filter-wrapper evolutionary approach for feature selection. *Memetic Computing*, *11*, 193–208.

Hassaine, S., Khomh, F., Guéhéneuc, Y.-G., & Hamel, S. (2010). IDS: An immune-inspired approach for the detection of software design smells. In *Proceedings of the 7th International Conference on Quality of Information and Communications Technology,* (pp. 343–348). IEEE.

Hawes, N. (2011). A survey of motivation frameworks for intelligent systems. *Artificial Intelligence*, *175*, 1020–1036.

He, H., & Garcia, E. A. (2009). Learning from imbalanced data. *IEEE Transactions on knowledge and data engineering*, *21*, 1263–1284.

Henderson-Sellers, B. (1995). *Object-oriented metrics: measures of complexity*. Prentice-Hall, Inc.

Holte, R. C., Acker, L., Porter, B. W. et al. (1989). Concept Learning and the Problem of Small Disjuncts. In *Proceedings of the 11th Joint international Conference on Artificial Intelligence* (pp. 813–818). Morgan Kaufmann.

Huang, C., Li, Y., & Yao, X. (2020). A survey of automatic parameter tuning methods for metaheuristics. *IEEE Transactions on Evolutionary Computation*, *24*, 201–216.

Jankowski, D., & Jackowski, K. (2014). Evolutionary Algorithm for Decision Tree Induction. In *Proceedings of the 13th Computer Information Systems and Industrial Management,* (pp. 23–32). Springer.

Karafotias, G., Hoogendoorn, M., & Eiben, Á. E. (2015). Parameter control in evolutionary algorithms: Trends and challenges. *IEEE Transactions on Evolutionary Computation*, *19*, 167–187.

Kessentini, M., Sahraoui, H., Boukadoum, M., & Wimmer, M. (2011). Search-Based Design Defects Detection by Example. In *Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering,* (pp. 401–415). Springer volume 6603.

Kessentini, W., Kessentini, M., Sahraoui, H., Bechikh, S., & Ouni, A. (2014). A Cooperative Parallel Search-Based Software Engineering Approach for Code-Smells Detection. *IEEE Transactions on Software Engineering*, *40*, 841–861.

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

Khomh, F., Vaucher, S., Guéhéneuc, Y.-G., & Sahraoui, H. (2009). A bayesian approach for the detection of code and design smells. In *Proceedings of the 9th International Conference on Quality Software,* (pp. 305–314). IEEE.

Khomh, F., Vaucher, S., Guéhéneuc, Y.-G., & Sahraoui, H. (2011). BDTEX: A GQM-based Bayesian approach for the detection of antipatterns. *Journal of Systems and Software*, *84*, 559–572.

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

Krętowski, M., & Grześ, M. (2005). Global learning of decision trees by an evolutionary algorithm. In *Information Processing and Security Systems* (pp. 401–410). Springer.

Langelier, G., Sahraoui, H., & Poulin, P. (2005). Visualization-based analysis of quality for large-scale software systems. In *Proceedings of the 20th IEEE/ACM international Conference on Automated*

software engineering, (pp. 214–223). ACM.

Lanza, M., & Marinescu, R. (2007). *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer Science & Business Media.

Liu, H., Liu, Q., Niu, Z., & Liu, Y. (2015). Dynamic and automatic feedback-based threshold adaptation for code smell detection. *IEEE Transactions on Software Engineering*, *42*, 544–558.

Luque, A., Carrasco, A., Martín, A., & de las Heras, A. (2019). The impact of class imbalance in classification performance metrics based on the binary confusion matrix. *Pattern Recognition*, *91*, 216–231.

Ma, C. Y., & Wang, X. Z. (2009). Inductive data mining based on genetic programming: Automatic generation of decision trees from data for process historical data analysis. *Computers & Chemical Engineering*, *33*, 1602–1616.

Maiga, A., Ali, N., Bhattacharya, N., Sabane, A., Gueheneuc, Y.-G., & Aimeur, E. (2012a). SMURF: A SVM-based incremental anti-pattern detection approach. In *Proceedings of the 19th Working conference on Reverse engineering,* (pp. 466–475). IEEE.

Maiga, A., Ali, N., Bhattacharya, N., Sabané, A., Guéhéneuc, Y.-G., Antoniol, G., & Aïmeur, E. (2012b). Support vector machines for anti-pattern detection. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering,* (pp. 278–281). IEEE.

Mansoor, U., Kessentini, M., Maxim, B. R., & Deb, K. (2017). Multi-objective code-smells detection using good and bad design examples. *Software Quality Journal*, *25*, 529–552.

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

Marinescu, R. (2002). *Measurement and Quality in Object Oriented Design*. Ph.D. thesis Politehnica University of Timisoara.

Marinescu, R. (2004). Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of the 20th IEEE International Conference on Software Maintenance* (pp. 350–359). IEEE.

Marinescu, R., Ganea, G., & Verebi, I. (2010). Incode: Continuous quality assessment and improvement. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering* (pp. 274–275). IEEE.

Martin, R. C. (2002). *Agile software development: principles, patterns, and practices*. Prentice Hall.

Moha, N., Gueheneuc, Y. G., Duchien, L., & Meur, A. F. L. (2010). DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Transactions on Software Engineering*, *36*, 20–36.

Mullick, S. S., Datta, S., Dhekane, S. G., & Das, S. (2020). Appropriateness of performance indices for imbalanced data classification: An analysis. *Pattern Recognition*, *102*, 107–197.

Murthy, S. K., Kasif, S., & Salzberg, S. (1994). A system for induction of oblique decision trees. *Journal of artificial intelligence research*, *2*, 1–32.

Murthy, S. K., Kasif, S., Salzberg, S., & Beigel, R. (1993). Oc1: A randomized algorithm for building oblique decision trees. In *Proceedings of AAAI* (pp. 322–327). Citeseer.

Obregon, J., Kim, A., & Jung, J.-Y. (2019). Rulecosi: Combination and simplification of production rules from boosted decision trees for imbalanced classification. *Expert Systems with Applications*, *126*, 64–82.

Oliveto, R., Khomh, F., Antoniol, G., & Guéhéneuc, Y.-G. (2010). Numerical signatures of antipatterns: An approach based on b-splines. In *Proceedings of the 14th European Conference on Software maintenance and reengineering,* (pp. 248–251). IEEE.

Ouni, A. (2014). *A Mono-and Multi-objective Approach for Recommending Software Refactoring*. Ph.D. thesis Faculty of arts and sciences of Montreal.

Ouni, A., Kessentini, M., Sahraoui, H., & Boukadoum, M. (2013). Maintainability defects detection and correction: a multi-objective approach. *Automated Software Engineering*, *20*, 47–79.

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

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

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

Palomba, F., Bavota, G., Penta, M. D., Oliveto, R., Lucia, A. D., & Poshyvanyk, D. (2013). Detecting bad smells in source code using change history information. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering* (pp. 268–278). IEEE Press.

Palomba, F., De Lucia, A., Bavota, G., & Oliveto, R. (2014). Anti-pattern detection: Methods, challenges, and open issues. In *Advances in Computers* (pp. 201–238). Springer.

Palomba, F., Panichella, A., De Lucia, A., Oliveto, R., & Zaidman, A. (2016). A textual-based technique for smell detection. In *2016 IEEE 24th international conference on program comprehension (ICPC)* (pp. 1–10). IEEE.

Palomba, F., Tamburri, D. A. A., Fontana, F. A., Oliveto, R., Zaidman, A., & Serebrenik, A. (2018c). Beyond technical aspects: How do community smells influence the intensity of code smells? *IEEE transactions on software engineering*, .

Palomba, F., & Zaidman, A. (2019). The smell of fear: on the relation between test smells and flaky tests. *Empirical Software Engineering*, (pp. 1–40).

Palomba, F., Zaidman, A., & De Lucia, A. (2018d). Automatic test smell detection using information retrieval techniques. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (pp. 311–322). IEEE.

Palomba, F., Zanoni, M., Fontana, F. A., De Lucia, A., & Oliveto, R. (2017). Toward a smell-aware bug prediction model. *IEEE Transactions on Software Engineering*, .

Pascarella, L., Palomba, F., & Bacchelli, A. (2019). Fine-grained just-in-time defect prediction. *Journal of Systems and Software*, *150*, 22–36.

Pecorelli, F., Palomba, F., Di Nucci, D., & De Lucia, A. (2019a). Comparing Heuristic and Machine Learning Approaches for Metric-Based Code Smell Detection. In *Proceedings of the IEEE/ACM International Conference on Program Comprehension* (p. 12). IEEE.

Pecorelli, F., Palomba, F., Di Nucci, D., & De Lucia, A. (2019b). Comparing heuristic and machine learning approaches for metric-based code smell detection. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)* (pp. 93–104). IEEE.

Ramirez, A., Romero, J. R., & Ventura, S. (2018). A survey of many-objective optimisation in search-based software engineering. *Journal of Systems and Software*, *149*, 382–395.

Rapu, D., Ducasse, S., Gîrba, T., & Marinescu, R. (2004). Using history information to improve design flaws detection. In *Proceedings of the 8th European Conference on Software Maintenance and Reengineering,* (pp. 223–232). IEEE.

Rasool, G., & Arshad, Z. (2017). A lightweight approach for detection of code smells. *Arabian Journal for Science and Engineering*, *42*, 483–506.

Sahin, D., Kessentini, M., Bechikh, S., & Deb, K. (2014). Code-Smell Detection as a Bilevel Problem. *ACM Transactions on Software Engineering and Methodology*, *24*, 1–44.

dos Santos Neto, B. F., Ribeiro, M., Da Silva, V. T., Braga, C., De Lucena, C. J. P., & de Barros Costa, E. (2015). AutoRefactoring: A platform to build refactoring agents. *Expert Systems with Applications*, *42*, 1652–1664.

Sharma, T., & Spinellis, D. (2018). A survey on software smells. *Journal of Systems and Software*, *138*, 158–173.

Sjoberg, D. I., Yamashita, A., Anda, B. C., Mockus, A., & Dyba, T. (2013). Quantifying the Effect of Code Smells on Maintenance Effort. *IEEE Transactions on Software Engineering*, *39*, 1144–1156.

Spadini, D., Palomba, F., Zaidman, A., Bruntink, M., & Bacchelli, A. (2018). On the relation of test smells to software code quality. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (pp. 1–12). IEEE.

Suen, C. Y. (1990). Recognition of totally unconstrained handwritten numerals based on the concept of multiple experts. In *Proceedings of the 1st International Workshop on Frontiers in Handwriting*

Recognition (pp. 131–143).

Suen, C. Y., Nadal, C., Legault, R., Mai, T. A., & Lam, L. (1992). Computer recognition of unconstrained handwritten numerals. (pp. 1162–1180). IEEE.

Tsantalis, N., & Chatzigeorgiou, A. (2009). Identification of Move Method Refactoring Opportunities. *IEEE Transactions on Software Engineering*, *35*, 347–367.

Tsantalis, N., & Chatzigeorgiou, A. (2011). Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software*, *84*, 1757–1782.

Tufano, M., Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., & Poshyvanyk, D. (2016). An empirical investigation into the nature of test smells. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 4–15). IEEE.

Van Rijsbergen, C. (1979). Information retrieval.

Vassallo, C., Grano, G., Palomba, F., Gall, H. C., & Bacchelli, A. (2019). A large-scale empirical exploration on refactoring activities in open source software projects. *Science of Computer Programming*, .

Vaucher, S., Khomh, F., Moha, N., & Guéhéneuc, Y.-G. (2009). Tracking design smells: Lessons from a study of god classes. In *Proceedings of the 16th Working Conference on Reverse Engineering,* (pp. 145–154). IEEE.

Vidal, S. A., & Marcos, C. A. (2012). Building an expert system to assist system refactorization. *Expert Systems with Applications*, *39*, 3810 – 3816.

Wickramarachchi, D., Robertson, B., Reale, M., Price, C., & Brown, J. (2016). HHCART: An oblique decision tree. *Computational Statistics & Data Analysis*, *96*, 12–23.

Wirfs-Brock, R., & McKean, A. (2003). *Object design: roles, responsibilities, and collaborations*. Addison-Wesley Professional.

Yamashita, A. (2012). *Assessing the capability of code smells to support software maintainability assessments: Empirical inquiry and methodological approach*. Ph.D. thesis Department of Informatics, Faculty of Mathematics and Natural Sciences, University of Oslo.

Yamashita, A., & Moonen, L. (2013a). Do developers care about code smells? an exploratory survey. In *2013 20th Working Conference on Reverse Engineering (WCRE)* (pp. 242–251). IEEE.

Yamashita, A., & Moonen, L. (2013b). Exploring the impact of inter-smell relations on software maintainability: An empirical study. In *Proceedings of the Conference on Software Engineering,* (pp. 682–691). IEEE.

Yamashita, A., & Moonen, L. (2013c). Exploring the impact of inter-smell relations on software maintainability: An empirical study. In *Proceedings of the Conference on Software Engineering,* (pp. 682–691). IEEE.

Zhang, D., & Tsai, J. J. (2003). Machine learning and software engineering. *Software Quality Journal*, *11*, 87–119.

Appendix A. Description of the handled code smells

In this research work, we considered eight code smell types described by Table 10, which are among the most studied anti-patterns in the software maintenance field (Fontana et al., 2012), (Fowler & Beck, 1999), (Lanza & Marinescu, 2007), (Martin, 2002), (Ouni, 2014), (Wirfs-Brock & McKean, 2003):

Table 10: List of commonly used code smells for the detection process in the literature

Code Smell / Antipattern	Description
God Class (aka Blob)	It appears when a large class centralizes most of the behavior of a system while other classes mainly include data.
Data Class	It occurs when a class stores data but not complex functionalities.
Feature Envy	It arises when a method calls much more methods from another class than their own.
Long Method	This smell refers to a method that has grown too long regarding lines of code.
Duplicate code	This smell happens when the same code structure is redundant in many classes.
Long Parameter List	This smell occurs when a method contains a long parameter list.
Spaghetti Code	This smell arises when the control structure of the code becomes complex and tangled.
Functional Decomposition	It happens when a class is conceived with the aim of performing a single function. Such bad practice is performed in the code by OO developers without experience.

Appendix B. Description of the used metrics

The used metrics through this work are presented in Table 11 (Chidamber & Kemerer, 1994),(Lanza & Marinescu, 2007),(Marinescu, 2002), (Ouni, 2014):

Table 11: List of the used metrics

Metric	Description
ANA - Average Number of Ancestors	This measure means the average number of classes from which every class inherits information.
AOFD - Access Of Foreign Data	This metric is used to count the attributes' number from unrelated classes, which are accessed directly or by invoking accessor (i.e. getter) methods.
CAM - Cohesion Among Methods of Class	This metric is used to compute the relatedness between methods of a class, calculated by applying the intersection of parameters of a method with the maximum independent set of all parameter types in the class.
CBO - Coupling Between Objects	It counts the number of classes that invoke a function or access a variable of a specific class.
CIS - Class Interface Size	It is used to count the number of existing public methods in a class. It is interpreted as the mean of the whole classes in a design.
CM - Changing Method	It counts the number of distinct methods, which call the measured method.
DAM - Data Access Metric	It is the ratio of the number of private (or protected) attributes to the total number of attributes declared in the class.
DCC - Direct Class Coupling	It counts the number of the various classes, and which class is directly related to. The metric contains classes, which are directly related by the attribute declarations and message passing (i.e. parameters) in the methods.
DSC - Design Size in Classes	This measure is used to count the total number of classes in the design without considering the imported library classes.
LOC - Lines of Code	It measures the size of a program by counting the number of the instructions in a class or method.
MFA - Measure of Functional Abstraction	This measure is the ratio of the number of methods inherited by a class to the entire number of methods reachable by member methods of the class.
MOA - Measure of Aggregation	This measure counts the number of data declaration where their types are classes defined by the user.
NOA - Number of Attributes	It is used to count the number of attributes for a given class in the program.

Metric	Description
NOAM - Number of Accessor Methods	This metric is used to count the number of accessor (i.e. getter) and mutator (i.e. setter) that belong to a given class.
NOF - Number of Fields	It counts the number of classes fields.
NOH - Number of Hierarchies	This metric is used to count the total number of class hierarchies in the design.
NOM - Number of Methods	This measure counts the number of methods defined by a class.
NOPA - Number of Public Attributes	It is used to count the number of public attributes for a given class in the program.
NPA - Number of Private Attributes	It is used to measure the number of private attributes for a given class.
TCC - Tight Class Cohesion	It is used to count the relative number of method pairs of a class that access in common at least one attribute of the measured class.
WMC - Weighted Methods per Class	This measure is used to compute the complexity of a class according to the number of existing methods in the class.
WOC - Weighted Of Class	This measure represents functional (i.e. non-accessor) methods in a class divided by the total number of members of the interface.

Appendix C. Motivations for the use of ODTs over axis-parallel trees in imbalanced classification

This appendix is devoted to motivate and justify the adoption of ODTs over axis-parallel trees for the case of imbalanced data. In fact, there are mainly three arguments that defend our choice (Das et al., 2018). First, ODT use oblique splitting hyper-planes that could be adapted to any geometrical shape of class boundaries, which is not the case of axis-parallel trees. Indeed, the latter could produce a considerable number of small disjuncts, which makes the classification task more difficult. Second, as illustrated by Figure 13, the use of oblique hyper-planes is more effective and efficient for an efficacious splitting of instances over classes. We observe that the axis-parallel tree needs 11 hyper-planes to separate the two classes, while the ODT requires only a single oblique hyper-plane that fits the boundary. This makes ODTs more efficient and even more effective than orthogonal trees. It is worth noting that an ODT could generate an orthogonal hyper-plane in addition to the oblique ones, when it assigns a unique feature a weighting

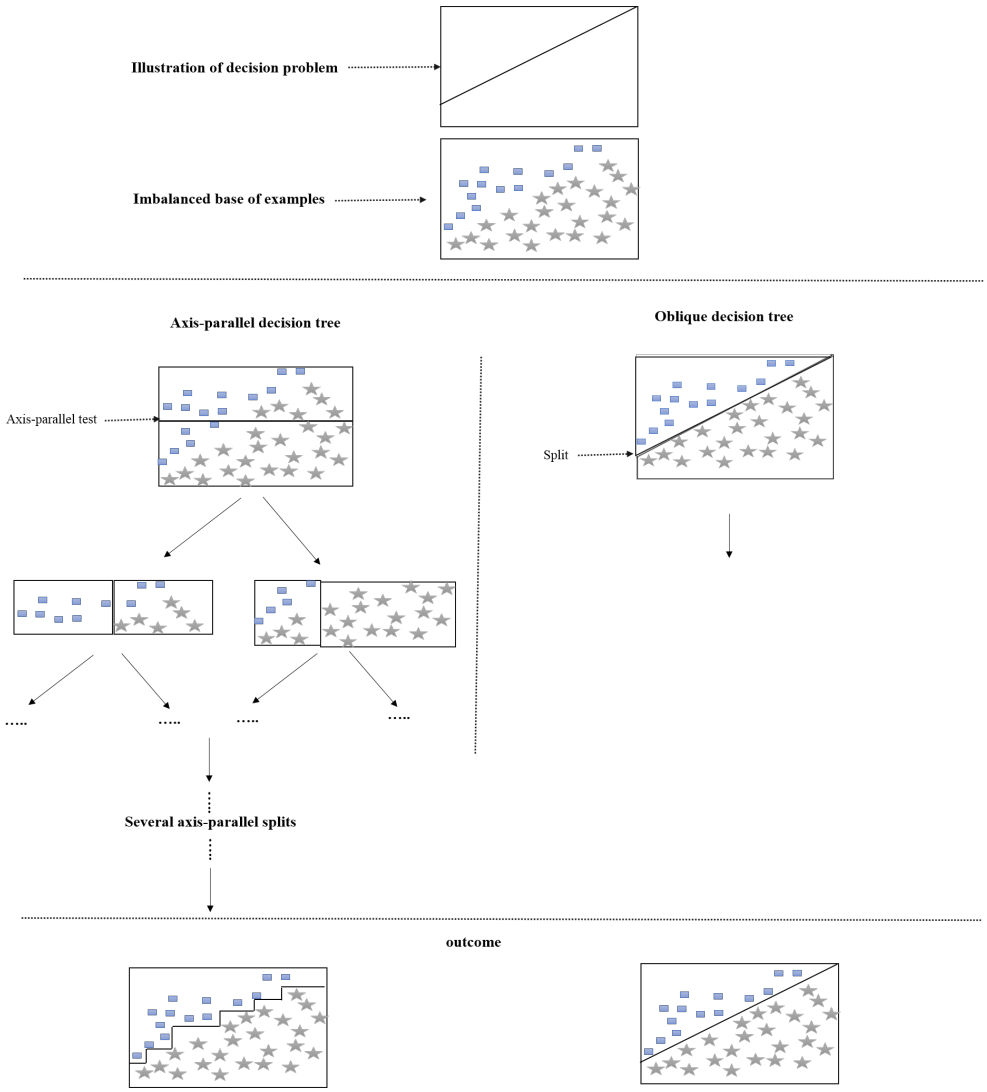


Figure 13: Illustration of the appropriateness of the ODT over the axis-parallel tree in imbalanced binary classification.

coefficient of one and all the other attributes are assigned null weights. We conclude that the axis-parallel tree could be a (rare) subcase of the ODT, which makes the latter more flexible with respect to the data splitting boundaries. Third, imbalanced data already contain small disjuncts, which correspond to small clusters of a particular class instances scattered over the feature space as illustrated by Figure 9. To detect and delimit these small disjuncts, it is wiser to use oblique splits instead of orthogonal ones as the former are more adapted to the geometrical form that could present the small clusters than the latter. Eventually, for the case of small disjuncts, the number of needed orthogonal hyper-planes is much higher than the one of oblique ones. Furthermore, axis-parallel trees could generate additional small disjuncts in such case, which makes the classification task more complex and difficult.

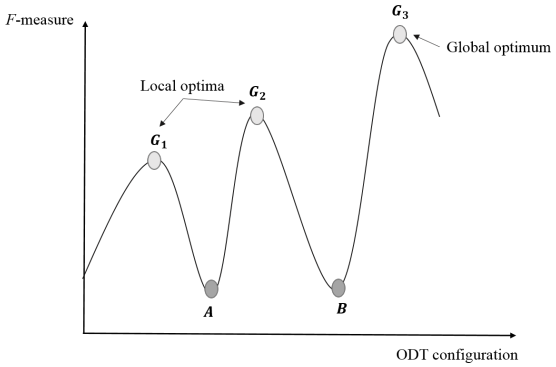


Figure 14: Illustrative example of two local optima and one global optimum for ODT configuration search based on the F-measure.

Appendix D. Evolutionary design versus greedy induction of ODTs

The goal of this appendix is to demonstrate how designing an ODT (or any kind of DTs) using EAs is better than inducing it using greedy machine learning algorithms (Barros et al., 2012; Al-Sahaf et al., 2019). Figure 14 presents a possible example of the F-measure function landscape. For simplicity of visualization, each point from the x-axis corresponds to an ODT configuration, while the y-axis presents the F-measure (configuration quality) value. Based on this figure, if the greedy algorithm starts from solution (configuration) A, in will converge to one of the two globally-optimal ODT structures G_1 or G_2. Similarly, if it starts the induction from solution B, it may either approximate the local optimum G_2 or the global one G3. Hence, the probability that a greedy ODT induction algorithm find a near globally-optimal configuration is very small, which is not the case of EAs thanks to two characteristics. On the one hand, the EA has a global search ability thanks to the evolution of a whole population (set of configurations) simultaneously instead of a single configuration, which is allows it to locate the promising regions in the search space (i.e., regions near G_1, G_2, and G_3). On the other hand, EAs has the ability to escape from local optima (e.g., solutions near G_1 and G_2) thanks to the probabilistic acceptance of worse configurations (of F-measure deterioration) using the binary tournament operator for mating selection. Indeed, to select $(N/2)$ children for reproduction, this operator performs $(N/2)$ iterations, where in each iteration two ODT configurations are randomly selected with replacement and the best one is saved in the mating pool. In this way, when selecting two ODT configurations as parents for crossover from the mating pool, these parents could contain deteriorated ODT configurations in addition to good ones. In this way, the EA allows the deterioration of the fitness function, which makes it able to: (1) escape local optima such as solution G_2 and then (2) direct the search towards the globally-optimal ODT configuration G_3.

Appendix F. ADIODE versatility with respect to metrics' definitions

The goal of this appendix is to show the insensitivity of ADIODE to the software quality metrics' definitions in code smells detection. Indeed, any metric could have more than one definition (expression) (Mäntylä & Lassenius, 2006) and this could be seen as one of the subjectivity facets of software engineers in defining not only quality metrics but also smell types. To assess the performance of ADIODE with other quality metrics' definitions, we have a conducted an experiment with a BE that is constructed using the inFusion tool [10]. This rule-based tool is adopted because its metrics' definitions are different from those used in our experimental study. Five smell types are considered that are Blob, Data Class, Feature Envy, Long Method, and Long Parameter List. Table 12 reports the F-measure and AUC values of ADIODE on the new BE (built with inFusion). We observe from this table that the performance of ADIODE is preserved. In fact, the F-measure lies within [87.02, 93.24] and the AUC varies within [0.8815, 0.9470]. These interesting results, with other metrics' definitions stemming from the inFusion tool, reveal that ADIODE performance is insensitive to the change of metrics' expressions. Indeed, it is the BE that depends on the metrics' definitions and not ADIODE itself. Thus, once the BE is build with the help of one or several tools in addition to the knowledge and experience of software engineers, ADIODE could be used to generated a set of optimized detectors whatever are the definitions of quality metrics. It is important to note that any software company has the freedom to choose any tools, metrics' definitions, and experts' opinions in building its base of smell examples. This motivates the need for ADIODE as a black-box tool that generates a set of high performing smell detectors based on the BE, independently of the tools and opinions employed in the development of this base.

Table 12: F-measure and AUC median scores of ADIODE

Code Smell	ADIODE	
	F-measure %	AUC
Blob (God Class)	93.24	0.9470
Data Class	90.20	0.9107
Feature Envy	89.15	0.8996
Long Method	87.02	0.8815
Long Parameter List	87.16	0.8830

[10] Available at http://www.intooitus.com/products/infusion