Empirical Software Engineering manuscript No.
(will be inserted by the editor)

Handling Uncertainty in SBSE: A Possibilistic Evolutionary Approach for Code Smells Detection

Sofien Boutaib · Maha Elarbi · Slim Bechikh · Fabio Palomba · Lamjed Ben Said ·

Received: date / Accepted: date

Abstract Code smells, also known as anti-patterns, are poor design or implementation choices that hinder program comprehensibility and maintainability. While several code smell detection methods have been proposed, Mantyla et al. identified the uncertainty issue as one of the major individual human factors that may affect developer's decisions about the smelliness of software classes: they may indeed have different opinions mainly due to their different knowledge and expertise. Unfortunately, almost all the existing approaches assume data perfection and neglect the uncertainty when identifying the labels of the software classes. Ignoring or rejecting any uncertainty form could lead to a considerable loss of information, which could significantly deteriorate the effectiveness of the detection and identification processes. Inspired by our previous works and motivated by the interesting performance of the PDT (Possibilistic Decision Tree) in classifying uncertain data, we propose ADIPE (Anti-pattern Detection and Identification using Possibilistic decision tree Evolution), as a new tool that evolves and optimizes a set of detectors (PDTs) that could effectively deal with software class labels uncertainty using some concepts from the Possibility theory. ADIPE uses a PBE (Possibilistic Base of Examples: a dataset with possibilistic labels) that it is built using a set of opinion-based classifiers (i.e., a set of probabilistic classifiers) with the aim to simulate human developers' uncertainty. A set of advisors and probabilistic classifiers are employed in order to mimic the subjectivity and the doubtfulness of software engineers. A detailed experimental study is conducted to show the merits and outperformance of ADIPE in dealing with uncertainty in code smells

Sofien Boutaib (Corresponding author)
SMART Lab, University of Tunis, ISG, Tunis, Tunisia. E-mail: boutaibsofien@yahoo.fr
Maha Elarbi
SMART Lab, University of Tunis, ISG, Tunis, Tunisia. E-mail: arbi.maha@yahoo.com
Slim Bechikh (Corresponding author)
SMART Lab, University of Tunis, ISG, Tunis, Tunisia. E-mail: slim.bechikh@fsegn.rnu.tn
Fabio Palomba
Software Engineering (SeSa) Lab, University of Salerno, Italy. E-mail: fpalomba@unisa.it
Lamjed Ben Said
SMART Lab, University of Tunis, ISG, Tunis, Tunisia. E-mail: lamjed.bensaid@isg.rnu.tn

detection and identification with respect to four relevant state-of-the-art methods, including the baseline PDT. The experimental study was performed in uncertain and certain environments based on two suitable metrics: *PF-measure_dist* (Possibilistic F-measure_Distance) and *IAC* (Information Affinity Criterion); which corresponds to the F-measure and *Accuracy* (*PCC*) for the certain case. The obtained results for the uncertain environment reveal that for the detection process, the *PF-measure_dist* of ADIPE ranges within [0.9047 and 0.9285], and its *IAC* lies within [0.9288 and 0.9557]; while for the identification process, the *PF-measure_dist* of ADIPE is in [0.8545, 0.9228], and its *IAC* lies within [0.8751, 0.933]. ADIPE is able to find 35% more code smells with uncertain data than the second best algorithm (i.e., BLOP). In addition, ADIPE succeeds to decrease the number of false alarms (i.e., misclassified smelly instances) with a rate equals to 12%. Our proposed approach is also able to identify 43% more smell types than BLOP and decreases the number of false alarms with a rate equals to 32%. Similar results were obtained for the certain environment, which demonstrate the ability of ADIPE to also deal with the certain environment.

Keywords Code smells · Subjectivity and doubtfulness of software engineers · Data labels uncertainty · Possibility theory · Possibilistic decision tree evolution · SBSE.

1 Introduction

Code smells are poor solutions to recurring implementation and design problems that may hinder the evolution of a system by making it hard for software engineers to carry out changes (de Paulo Sobrinho et al. 2018; Sharma and Spinellis 2018). As the introduction of these smells is usually unavoidable (Tufano et al. 2017), they should be detected and then refactored as early as possible (Mansoor et al. 2013). To ease the detection step, Fowler and Beck (1999) have defined the main symptoms of each smell type as well as its corresponding refactoring solution. Several code smell detection approaches have been proposed in the literature that could be classified into three categories: (1) rule-based approaches (Lanza and Marinescu 2007a), (2) machine learning-based approaches (Fontana et al. 2016b; Pecorelli et al. 2020b), and (3) search-based ones (Kessentini et al. 2011). A summary of notable recent software-based approaches (rule-based, machine learning-based, and search-based approaches) are summarized in Table 1. Rule-based approaches use a set of predefined rules that are based on the combination of metrics and thresholds; by design, they suffer from the problem of threshold calibration that has been often reported to lead software developers ignoring the code smell instances they output (Fontana et al. 2016a; Palomba et al. 2017a). Different from the rule-based approaches, the machine learning-based approaches build a classifier model using a base of smell examples; unfortunately, the model building process is greedy and hence locally-optimal classifiers are obtained (Barros et al. 2012; Al-Sahaf et al. 2019). Search-based approaches evolve a set of detection rules using also a BE and usually exploit Evolutionary Algorithms (EAs) as search engines because of their ability to find (near) globally-optimal detectors (Fernandes et al. 2016; Di Nucci et al. 2018).

In fact, the software engineer intervenes in all the previously mentioned approaches either by defining the rules of rule-based approaches, or by constructing the BE

Table 1 Comparison between the proposed approaches for the detection of code smells.

Category	Approaches	Main idea	Number of Considered Smells	Data Imbalance Consideration	Ability to Deal with Small-Disjuncts	Ability to Deal with Data Overlap	Uncertainty
Rule-based approaches	DECOR (Moha et al. 2010)	Detecting code smells based on a set of predefined rules.	3	No	No	No	No
	JDeodorant (Tsantalis and Chatzigeorgiou 2009)		3	No	No	No	No
Machine learning -based approaches	DT Amorim_2015	Detecting code smells using a classifier model built based on a Base of Examples (BE).	4	No	No	No	No
	SVMDetect (Maiga et al. 2012a,b)		3	No	No	No	No
	(Baseline) PDT Jenhani_Dissertation)	Detecting code smells using a possibilistic classifier model ~generated based on a Possibilistic ~Base of Examples (PBE).	8	No	No	No	Yes
Search-based approaches	GP (Ouni et al. 2013)	Evolving a number of rules in the form of IF-THEN generated from a BE based on the genetic programming.	8	No	No	No	No
	MOGP (Mansoor et al. 2017)	Generating IF-THEN rules using a multi-objective method (NSGA-II) ~through the optimization of two competing objectives.	8	No	No	No	No
	BLOP (Sahin et al. 2014)	Modeling the problem of code smell detection as a bi-level one to address ~the lack of diversity that might have a BE.	8	No	No	No	No
	ADIPOK (Boutaib et al. 2021)	Detecting and identifying code smells based on a possibilistic K-NN obtained based on a PBE.	8	Yes	No	No	Yes
	ADIPE (Our work)	**Detecting and identifying code smells using an evolved possibilistic classifier model generated based on a PBE.**	**8**	**Yes**	**Yes**	**Yes**	**Yes**

of machine learning-based approaches or search-based ones. Although most of the existing detectors, especially those included in the latter two categories, have shown promising detection capabilities, they still share a critical limitation that reduces their usage in practice. In fact, the existing detectors do not take into account the subjectivity of the human experts. These latter suppose that the software engineers express certain decisions regarding the smelliness of the software classes (i.e., the considered decisions are taken in a certain environment). However, according to recent findings in this research field (Yamashita and Moonen 2013; Palomba et al. 2014; Taibi et al. 2017), the outputs of the automated detectors (i.e., code smells) are subjectively perceived by developers based on their knowledge and expertise. This is mostly due to the doubtfulness in the decisions taken by the software engineers regarding the smelliness of software classes and the identification of the existing smell types (Taibi et al. 2017). Therefore, the software engineers are uncertain when expressing their decisions (i.e., the considered decision are subjectively taken in an uncertain environment). One can notice from Table 1 that all the existing approaches do not consider the uncertainty data problem. The underestimation of this problem by software engineering developers may cause a problem in the applicability of code smell detection and identification. For instance, Fontana et al. (2016a) identified a number of "conceptual" false positive instances given by existing detectors. These latter are able to identify code smells that are not perceived by developers as actual design issues and that are unavoidably ignored in practice. As shown in previous work (Mantyla et al. 2004), two developers could perceive different smell types over a software class. This subjectivity and uncertainty could be explained by the fact that they could have different experience degrees. Therefore, a class could be Smelly (i.e., includes a smell type) by one developer and Non-smelly by another expert. To mimics such a situation, we have chosen two different advisors (PMD and JDeodorant), which simulate the expert opinions for the detection of the Long Method smell type. As shown in Figure 1, two advisors could have different opinions (i.e., subjectivity) regarding the same method as they are implementing different detection rules. Moreover, the sources of subjectivity are the human developers since these latter are only able to express their uncertainty using likelihood values. Moreover, the sources of subjectivity are the human developers since these latter are only able to express their uncertainty using likelihood values. Existing works have demonstrated that the likelihood values (i.e., probability theory) are not effective for uncertain data (Jenhani 2010). To alleviate such a problem, we propose to employ the possibility theory, which has proven its adequacy to the uncertain data case (Jenhani et al. 2008b).

The solution to this problem can be modeled as an **uncertain class label classification** (Tsang et al. 2009). In this setting, a dependent variable (in our case, the smelliness of a class), is not modeled as a binary distribution (i.e., presence/absence of a smell) but with multiple values forming a probability that indicates the likelihood that a developer may consider a class belonging to one of the two categories (i.e., smelly/non-smelly). More precisely, in a typical use case scenario involving uncertain data, software practitioners can either reject the suggestion given by a detector or replace it with crisp data (i.e., certain data). Such practice is done *a-posteriori*, namely when the code smell candidates have been already presented to developers: this does not allow benefiting from the maximum amount of information hidden in the data

```
        protected String rtrim(String s)
        {
        // if the string is empty do nothing and return it
        if ((s == null) || (s.length() == 0))
        {
        return s;
        }
        // get the position of the last character in the string
        int pos = s.length();
        while ((pos > 0) && Character.isWhitespace(s.charAt(pos - 1)))
        {
         --pos;
        }
        // remove everything after the last character
        return s.substring(0, pos);
        }
    }
```

Fig. 1 Illustration of the Long Method smell type using the rtrim () method of the Apache Common CLI (Foundation 2004). This method is considered as a Long method using the JDedoroant advisor, while it is a normal method using the PMD.

(Jenhani et al. 2008b) and consequently it may deteriorate the overall performance of the detectors. Data mining and machine learning researchers have proposed a number of uncertainty theories and one of them is the **possibility theory**. In this paper, we propose ADIPE as a new tool to *detect* and *identify* code smells under uncertainty. The detection problem consists of discriminating whether a certain class is affected by a generic code smell, while the identification problem aims at pointing out the existence of a particular type of code smell (e.g., a God Class). The flowchart of our proposed approach in an uncertain environment is illustrated in Figure 2. One can see from Figure 2 that we can identify three phases in our proposed approach that are: (1) the construction of the BE phase, (2) the training phase, and (3) the application phase. In the construction of the BE phase, we can identify two possible scenarios: (1) the engineer is certain about the smelliness of a given software class or (2) the engineer has doubtfulness (uncertainty) about the smelliness of a given software class. In the first scenario, the software engineer does not has doubtfulness in determining the smelliness of a particular software class for this reason, the values of the software class labels are set to 0 or 1. In the second scenario, the software engineer is not certain about the smelliness of the software classes, therefore, the class labels values are expressed as likelihood values. We note that the BE has the form of a matrix where the lines are the Object Oriented (OO) software classes and the colons are the metrics values, while the last colon corresponds to the labels of the software classes (i.e., smelly or non-smelly for the case of detection or the smell type for the case of identification). In fact, our algorithm merges the opinions of human experts expressed in the form of probability values based on the voting fusion in order to obtain a single decision. Since the probability theory is unable to handle the subjective piece of information [2'], we have converted the resulted probability distributions over the software class labels into possibility ones. Motivated by the interesting performance of the PDT (Jenhani et al. 2008b), which is a combination between Possibility theory and decision tree, we have proposed to evolve a set of PDTs using the GA (Genetic Algorithm) metaheuristic. Therefore, during the training phase, ADIPE will use the

constructed BE (certain or uncertain BE) to outcome with a set of optimized detectors (PDT ensemble) that will be used by the software engineer to detect the smelliness of any unseen (i.e., unlabeled) OO software class in the application phase. Generally speaking, the unseen software classes of a targeted software project can be predicted based on the software classes that have been trained during the training phase and its labeled instances (Zimmermann et al. 2009; Hosseini et al. 2018). It is worth noting that the uncertainty could appear in the identification case. However, in such a case, the output of ADIPE will be a set of specific optimized smell detectors for each considered smell type.

As ADIPE combines several techniques from the computational intelligence field, it is important to justify our choices:

1. The suitability of PDTs to deal with the uncertain data classification problem (Jenhani et al. 2008b).
2. The capability of PDTs to learn from a training set characterized by uncertain class labels represented by possibility distributions (Jenhani et al. 2008b).
3. The ability of GAs to avoid the local optima in the PDT search space; which is not the case for state-of-the art greedy machine learning algorithms for PDT induction (cf. Appendix F);
4. The solid structure of smell detectors as PDTs; which is not the case of most existing search-based detection approaches that evolve a set of ad-hoc rules; and
5. The adaptive fusion of possibilistic distributions using conjunctive and disjunctive aggregations; which facilitates the decision making whatever are the detectors' states (in a concordance or discordance (Dubois and Prade 1994a)).

The performance of our ADIPE approach is assessed based on a detailed empirical study involving six well-known open-source projects submerged by uncertainty. The comparison is made with respect to four state-of-the-art baseline approaches. The experimental results of our study reveal that ADIPE outperforms its competitors in all the considered software projects under an uncertain environment. For the detection process, ADIPE succeeds to find 35% more code smells with uncertain data than BLOP. Moreover, ADIPE succeeds to decrease the number of false alarms with a rate equals to 12%. ADIPE has demonstrated its ability to identify 43% more smell types than BLOP and it decreases the number of false alarms with a rate equals to 32%. To sum up, the main contributions of this paper are:

1. Building a new base of possibilistic smell examples in the aim to aggregate the various expert engineers' subjective opinions, in which the expert quantify its uncertainty through elements from the possibility theory;
2. Proposing ADIPE as a new method and tool to detect and identify code smells under an uncertain environment;
3. Showing the performance of our ADIPE method on a set of detailed and statistically analyzed comparative experiments on six well-known open source projects with respect to four existing recent and prominent works in addition to the baseline PDT (Jenhani et al. 2008b).

It is important to note that the use of the possibility distribution concept in smells detection is not new, as we have already employed it in our previous work ADIPOK

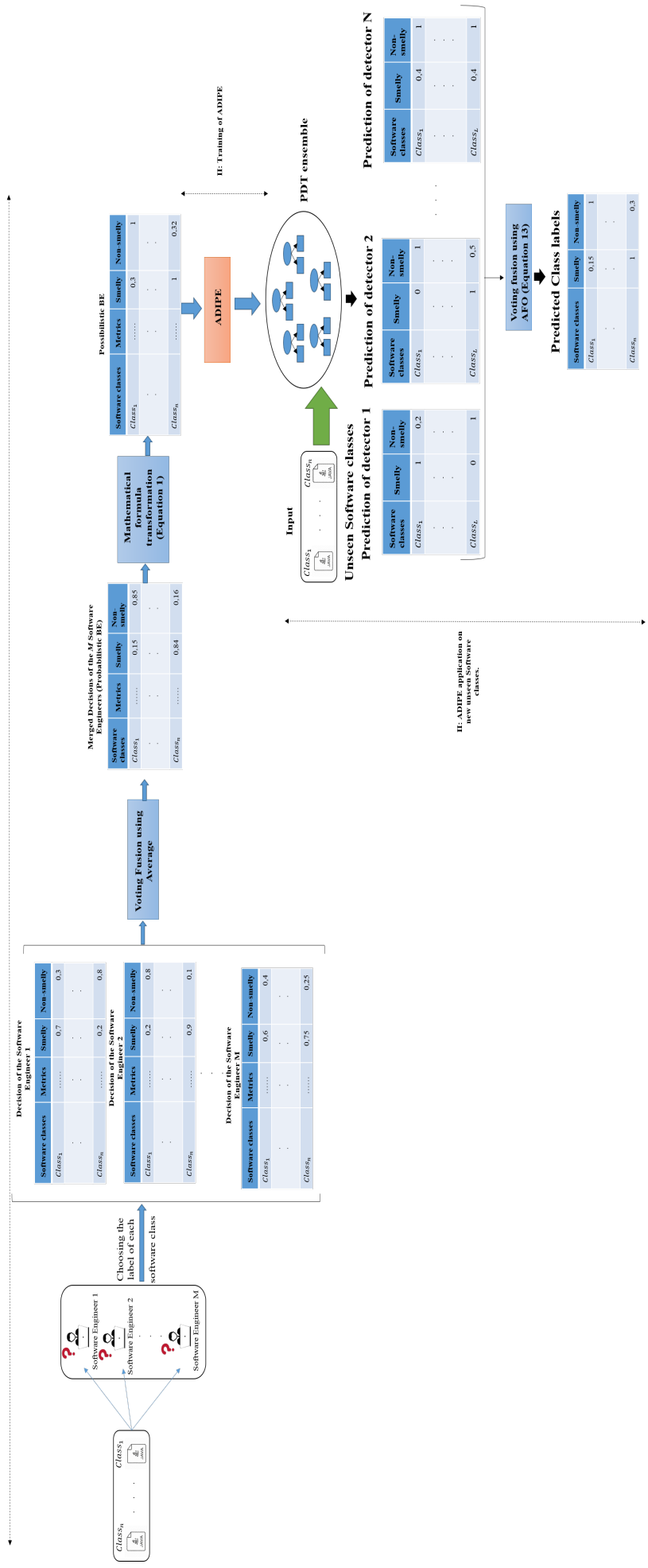


Fig. 2 Illustrating the main framework of our proposed algorithm in the uncertain environment. The certain case is a sub-case of the uncertain one with a two binary distributions composed of 0 and 1. The detection case is performed in a within-project manner where the unseen OO classes are labeled based on the trained OO classes of the source projects considered in the training phase.

(Boutaib et al. 2021). This latter is an evolutionary algorithm that evolves a population of PK-NNs (Possibilistic K-NNs). The main limitations of this tool are: (1) the definition of the number of nearest neighbors K, which heavily influences the results (cf. Figure 6); and (2) the inability to deal with small disjuncts and data overlap (no use of splitting hyper-planes)(cf. Figure 3). The main differences between ADIPOK and ADIPE can be summarized as follows (cf. Table 1):

1. The adoption of the PDT instead of the PK-NN as baseline classifier for evolution based on the *PF-measure_dist*;
2. Showing the advantages of the classifier change in solving two main challenges in smells detection that are related to the data imbalance: (a) small disjuncts and (2) data overlap (cf. Figure 4);
3. In addition to demonstrating the role of the non-possibility degree X in finding more effective splitting hyper-planes (cf. Figure 5), X values are shown to define the prioritization values of software classes for refactoring.
4. Extending the experiments using other existing performance metrics, notably the precision and the recall; which allows a better positioning of ADIPE with respect to the related works

Through the first section of this paper, we have described the problem statement of this paper by exposing one of the most influential human factors in code smells detection, which is uncertainty in software classes labelling. We have also pointed out that the main sources of uncertainty are the developers' subjectivity, doubtfulness, and personal experience. This uncertainty issue was already discussed by some researchers since more than fifteen years such as Mantyla et al. (2004), but unfortunately it was usually omitted so far by the SE (Software Engineering) community, including the SBSE one. The second section introduces some important concepts related to the possibility theory that will aid the reader to understand the rest of the paper. The third section is devoted to describe ADIPE as a new tool that is able to process developers' uncertainty related to software classes' labels. ADIPE uses a PBE to build a set of smells detectors each encoded as PDT. As the specialized literature did not consider the uncertainty issue in building the BE, we have constructed a PBE that mimics the developers' uncertainty by following a three-step process: (i) first, a set of advisors are used to generate different certain BEs; (ii) then a set of probabilistic classifiers are applied to transform the labels values into probability distributions; (iii) finally these distributions are converted into possibility ones. It is worth noting that the opinion-based classifiers (i.e., probabilistic classifiers) are used to simulate the human experts' opinions by generating probability distributions for the classes of the software projects (Bounhas et al. 2014). The ADIPE tool communicates with the PBE when evaluating each PDT (detector) using the *PF-measure_dist* (Possibilistic version of the *F-measure*). The latter corresponds to the *F-measure* in the certain case.

After detailing the solution encoding and the genetic operators of ADIPE, a detailed comparative experimental study is conducted in the fourth section for both cases: (i) the uncertain case and (ii) the certain one. The experiments are conducted in a within-project manner by following the hold-out validation method (70% of the data is for training and the remaining 30% is for performance testing). The experimental results reveal that ADIPE remarkably surpasses its competitors for the detection and

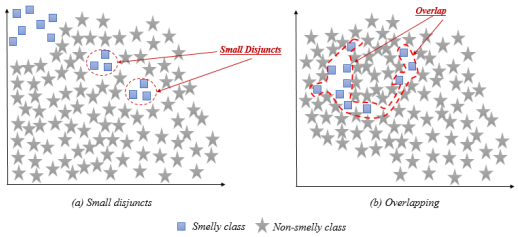


Fig. 3 Illustration of the difficulties encountered when handling imbalanced data: (a) small disjuncts and (b) overlapping.

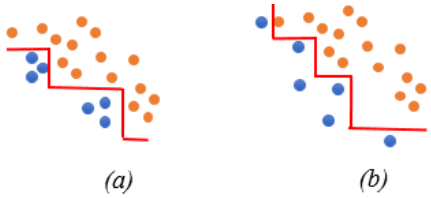


Fig. 4 Examples of effective hyper-planes that allow dealing with small disjuncts and overlapping data

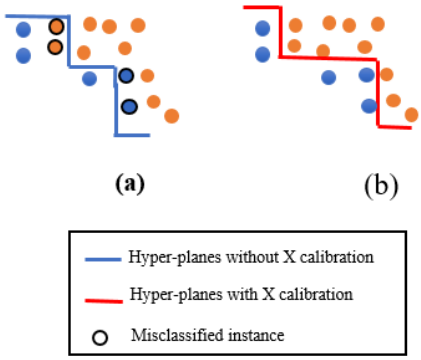


Fig. 5 The effect of the X values in the calibration of the PDT splitting hyper-planes based on ADIPE fitness function.

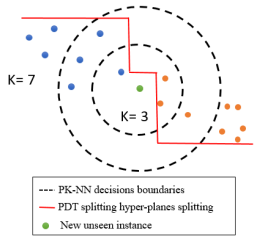


Fig. 6 The difference between the splitting hyper-planes of ADIPE and ADIPOK ones.

identification cases in both uncertain and certain environments. The fifth section discusses the various threats that could affect the validity of our experimentations. The sixth section summarizes the related work and the main characteristics of existing related works. The last section concludes the paper and gives some avenues for future research. It is important to note that a set of appendices are presented at the end of the paper to give some background concepts and details that could be valuable for the interested reader.

2 Background

Possibility theory is one of the theories that offer a simple model to deal with an uncertain environment Zadeh (1978); Dubois and Prade (1988). Let Ω be the universe of discourse for the different states of knowledge $\{\omega_1, \omega_2,..., \omega_n\}$. Each state of knowledge (i.e., ω_i) corresponds to a class label in the BE. In other words, the universe of discourse for the case of detection is $\Omega_{detection}$={ Smelly, Non-smelly}, while for the case of identification the universe of discourse corresponds to $\Omega_{identification}$={ Blob, Data Class, Long Method, Long Parameter List, Duplicate Code, Feature Envy, Spaghetti Code }. The possibility distribution, denoted by π, is a function that maps ω from Ω to obtain a possibility degree from the unit interval (i.e., [0, 1]), which is a representation of the real-world knowledge of the software engineer. $\pi(\omega_k)$ = 1 means that the realization of ω_k is completely possible, while $\pi(\omega_k) = 0$ means that ω_k is a rejected state. Therefore, $\pi(Smelly) = 1$ means that the passed software project class is completely possible smelly, while $\pi(Smelly) = 0$ means that the given software project class is completely non-smelly. Moreover, $\pi(\omega_k) > \pi(\omega_j)$ means that ω_k is more plausible (or is more specific) than ω_j. In order to measure, the degree of conflict between two sources of information, we use The Inconsistency measure (Inc). For more details about the computation of this measure, the reader can refers to Appendix C.

In real world, the information could be provided by different sources. The sources of information in the software maintenance domain are software engineers where the information corresponds to their opinions (possibility distributions). These latter should be combined to provide useful information. However, the choice of the combination modes depends on the state of sources (i.e., if they are in agreement or not). Two modes of combination of the source of information have been employed in the literature: The conjunctive fusion Dubois and Prade (2000) and the disjunctive one Dubois and Prade (2000). More details about these two fusion modes are given in Appendix C.

Tending to compare two distributions, some measures are proposed in possibility theory such as Minkowski distance (Dunford et al. 1971), information closeness (Higashi and Klir 1983), Sanguesa et al distance (Sangüesa et al. 1998), and information divergence (Kroupa 2006). Nevertheless, the previously mentioned measures of similarity cannot satisfy the inconsistency criterion, which is considered as one of the important criteria for measuring possibilistic similarity. The information affinity (denoted by *Aff*) (Jenhani et al. 2007) is a measure that was conceived to compute the amount of closeness between two opinions (i.e., possibility distributions).

This latter takes into account the distance and the inconsistency criteria and it is defined as follows:

$$Aff(\pi_1, \pi_2) = 1 - \frac{\kappa * d(\pi_1, \pi_2) + \lambda * Inc(\pi_1, \pi_2)}{\kappa + \lambda} \quad (1)$$

where $\kappa > 0$ and $\lambda > 0$ are two parameters defined by the user. In fact, κ and λ parameters should be equal to give similar importance for both distance and inconsistency criteria. $Inc(\pi_1, \pi_2)$ refers to the inconsistency (cf. Equation 17) while $d(\pi_1, \pi_2)$ refers to the normalized Manhatten distance. For more details about the information Affinity and the manhatten istance, the readers can refer to Appendix C.

3 Anti-pattern detection and identification using possibilistic decision tree evolution: ADIPE

In this section, we describe and detail the main components of our approach, denoted as ADIPE, for code smell detection and identification under an uncertain environment. The uncertainty is mainly inherent in the class labels due to the subjectivity and the uncertainty of the software engineers' opinions about the smelliness of a software class and the types of the existing smells. Indeed, ignoring the uncertainty could cause the deterioration of the quality of the generated results by the detector. We notice that both tasks are done using the same mechanism, but using different BEs with uncertain class labels. The class labels are assigned with likelihood values that represent the uncertainty of the expert opinions called possibility degrees. These latter are constructed using five probabilistic classifiers (Naïve Bayes classifier (Friedman et al. 1997), Probabilistic K-NN (Holmes and Adams 2002), Bayesian Networks (Pearl 1982, 1985), Naïve Bayes Nearest Neighbor (Behmo et al. 2010), and Probabilistic Decision Tree (Quinlan 1987)) to assign (predict) aggregated probability distribution for each existing software class in the BE. The use of probabilistic classifiers is extended from the idea suggested in (Bounhas et al. 2014), in which a number of classifiers are used to simulate the experts. In this work, the selected classifiers are used in the aim to simulate the uncertainty of the experts (i.e., software engineers) to label the BE software classes. This will help the expert to quantify its uncertainty about the smelliness of a software class in the form of likelihood values.

Afterward, these generated values are converted into possibility distributions through a conversion formula (cf. Equation 2). Regarding the detection case, the base could include several smell types or the opposite; while for the identification case, the base encompasses only a single smell type. That is why the identification is considered as a special case of the detection task in which the approach runs only on a single smell type. To facilitate the comprehension of the ADIPE working principle, we first present the main motivations behind its design. Second, we detail the artificial construction of the PBEs. Third, we illustrate the global schema of our detection method in an uncertain environment. Fourth, we specify how the detectors (i.e., PDTs) are evolved using the GA. We note that the PDTs are adopted in our work since they are suitable to learn from instances characterized with uncertain class labels. In the rest of this section, we describe the solution encoding, the fitness function, and the crossover and mutation operators. Finally, we illustrate how the obtained detectors could be employed for the detection and identification of code smells.

3.1 PBE: Constructing the dataset with possibilistic labels

Like many data mining fields, the SE industry could be susceptible to the uncertainty issue. The BEs could be submerged by uncertain class labels. The main uncertainty sources could be related to: (1) the lack of human experts' knowledge and/or (2) the subjectivity of their possibly conflicting opinions. In such an uncertain environment, the expert could express their opinion in the form of likelihood degrees each referring to the membership degree of every software class to every class label. To be able to generate possibilistic smell detectors, we need a set of PBEs. In such PBEs, each software class (i.e., instance) could be assigned a set of class labels each having a Possibility degree. Unfortunately, existing BEs do not contain uncertain class labels that could serve as training datasets for possibilistic detectors induction. As the goal of this paper is to process uncertainty in only class labels and not in features, only the class labels will be substituted with possibility distribution; while the original features' values are preserved. The possibility distributions are generated through the opinion-based classifiers (extended from (Bounhas et al. 2014)) and the conversion formula (Dubois and Prade 1985) (cf. Figure 7). The transformation process is described as follows:

1. First, we employ five different advisors (DECOR (Moha et al. 2010), JDeodorant (Tsantalis and Chatzigeorgiou 2009), inFusion[1], iPlasma[2], and PMD (Gopalan 2012)) to generate five different crisp BEs. We notice that the BE includes all the OO classes of the chosen software systems in this work (see Table 2).
2. Second, we execute independently five probabilistic classifiers (previously mentioned) on each of the five obtained crisp BEs. Then, we apply the voting fusion using the average operator in two times. In the first time, the voting fusion is employed to aggregate the obtained probabilistic BE for each classifier and in the second time, it is employed to obtain a single probabilistic BE (cf. Figure 7).
3. Finally, the obtained probability distributions are transformed into a possibility distributions using the following conversion formula, proposed by (Dubois and Prade 1985):

$$\pi(\omega_i) = i * p(\omega_i) + \sum_{j=i+1}^{n} p(\omega_j), \forall i = 1..n \tag{2}$$

where the probability distribution p defined on Ω should be sorted in descending order $(p(\omega_1) \geqslant p(\omega_2) \geqslant ... \geqslant p(\omega_n))$ before starting the transformation of p into π. Moreover, the sum of the probability distribution degrees should be equal to one. According to Equation 2, the distribution value will have one of the following forms: [X Y] or [Y X] where X is in [0 1] and Y is equal to 1.

3.2 Basic schema and fitness function

Figure 8 presents the main schema of the ADIPE method, which is mainly composed of two modules: (1) the possibilistic smell detectors generation module

[1] http://www.intooitus.com/products/infusion

[2] http://loose.upt.ro/iplasma/

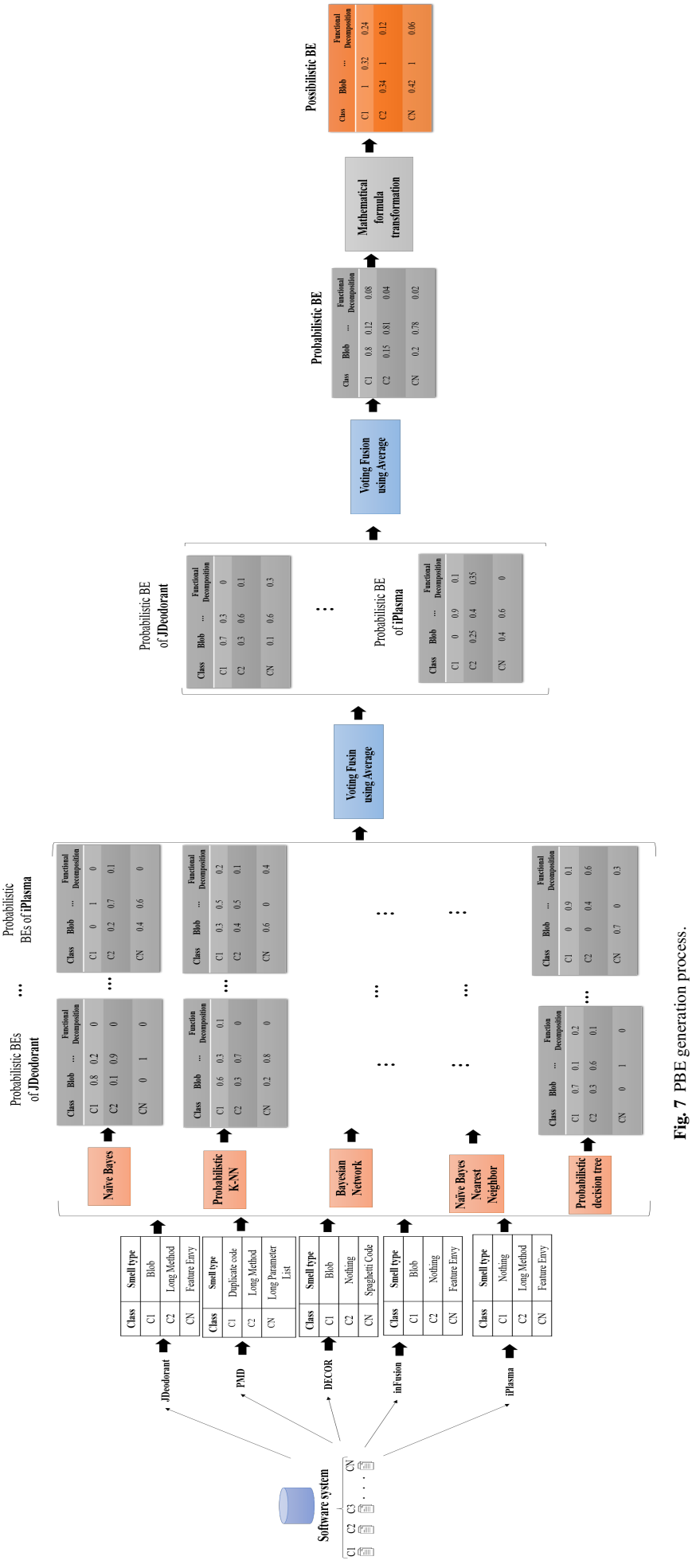
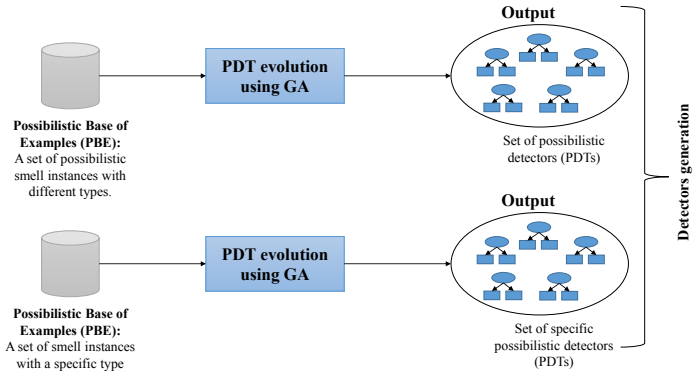
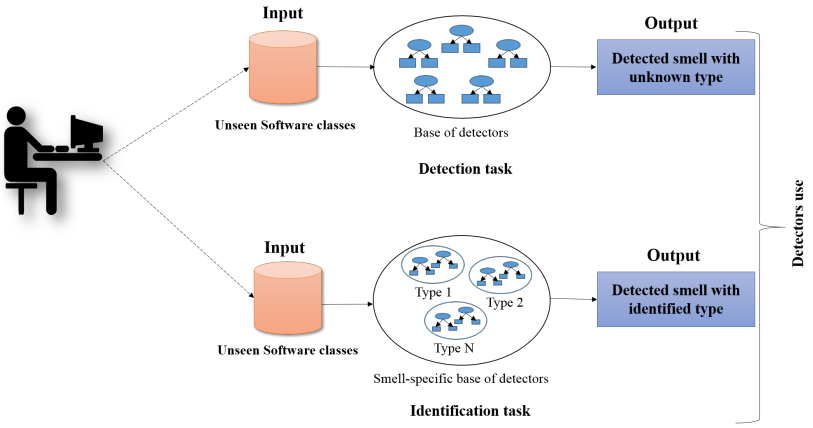


Fig. 7 PBE generation process.



(a) Possibilistic smell detectors generation



(b) Detection and Identification tasks

Fig. 8 The global schema of the ADIPE approach.

and (2) the possibilistic smell detectors application one. The former produces a set of optimized PDTs, while the latter applies these generated PDTs on unseen software classes. It is important to note that It is important to note that the possibility distributions obtained from the generated PDTs are aggregated using the Adaptive Fusion Operator (AFO) to produce a single possibility distribution degree (Dubois and Prade 1994a). Moreover,over the detection process, all the produced detectors are gathered together into only one BE, while each kind of smell type is collected into a specific base through the identification process.

Each individual (PDT) of the GA's population should be assessed to determine the solution quality after the population initialization process as well as at any time a new offspring individual (PDT) is created by applying the crossover and/or the mutation

operators. Based on these assigned fitness (quality) values, the selection process is performed to keep the chosen individuals for the next generation. Many works have presented different fitness functions as part of evolutionary classification among them the *accuracy* metric in (Ma and Wang 2009). However, the adoption of the existing metrics as a fitness function in the case of uncertain class labels represents a crucial issue. As we mentioned previously, the PDT generates possibility distributions during the classification process. The existing metrics will assign the classes. Hence, these metrics give much attention to only the most plausible class labels while the rest are ignored. For instance, suppose that we have the following possibility distribution for three smell types in the case of identification: $\pi(Blob) = 1$, $\pi(Long\ Method) = 0.6$, $\pi(Duplicate\ Code) = 0.8$. The existing metrics consider the Blob smell as it is the fully possible smell, while the remaining ones are discarded. However, according to the presented example, the remaining smell types have a high possibility to exist in the given software class. In fact, ignoring some possibility degrees could cause a loss of an important amount of information that is generated by detectors. As a result, inadequate or missing refactoring methods for the correction of code smell may lead to the deterioration of the software performance.

The code smell detection is considered as an imbalanced binary classification problem (Pecorelli et al. 2019, 2020a) where the BE is composed of two sub-sets: (1) the majority class and (2) the minority one. The former cardinality is much more than the latter one, which causes an imbalance problem that should be managed by the detector. The high imbalance ratio remarkably appears in the identification task as this latter is similar to the detection task but with a single smell type. To handle the data imbalance issue, our modeled fitness function is based on the F-measure classification metric (Van Rijsbergen 1979). Our choice could be explained by the fact that the F-measure relies on the harmonic mean of precision and recall over the minority class; this renders its insensitive to whenever the data imbalance ratio is, which is not the case for the existing software-based approaches Table 1. Unlikely, the F-measure is not suitable for the uncertain environment and hence it will deteriorate the detector's performance. Motivated by this observation, we have developed a new metric called *PF-measure_dist* (Possibilistic *F-measure*_distance) that is expressed by equation 3. The *PF-measure_dist* considers the average distances between (π^{res}) and the initial possibility distribution (π^{init}) of each classified unseen (unlabeled) software class $\overrightarrow{I_j}$. The proposed measure calculates the amount of closeness between the predicted software class comparing to the label of its corresponding one in the ground truth. When *PF-measure_dist* is close to 1, the resulting detector is accurate and the resulting possibility distributions are of high-quality as well as faithful compared to the real (initial) ones. This means that the predicted possibility distribution is very close to the ground truth, more precisely to the opinion of the software engineer. In contrast, the detector is considered extremely bad in the case where its assigned fitness function relying on the *PF-measure_dist* metric falls to 0, which means that the generated possibility distribution for a given software class is far from the software engineer's opinion. The *PF-measure_dist* is expressed as follows:

$$PF - measure\_dist = 2 \times \frac{Precision_{\_dist} \times Recall_{\_dist}}{Precision_{\_dist} + Recall_{\_dist}} \tag{3}$$

$$Precision\_dist = \frac{TP\_dist}{TP\_dist + FP\_dist} \tag{4}$$

$$Recall\_dist = \frac{TP\_dist}{TP\_dist + FN\_dist} \tag{5}$$

$$TP\_dist = \sum_{\overrightarrow{I_j} \in \text{ASCcc}} NSD(\overrightarrow{I_j}) \tag{6}$$

$$TN\_dist = \sum_{\overrightarrow{I_j} \in \text{ANSCcc}} NSD(\overrightarrow{I_j}) \tag{7}$$

$$FP\_dist = \sum_{\overrightarrow{I_j} \in \text{ASCmNs}} NSD(\overrightarrow{I_j}) \tag{8}$$

$$FN\_dist = \sum_{\overrightarrow{I_j} \in \text{ANSCms}} NSD(\overrightarrow{I_j}) \tag{9}$$

$$Sd(\overrightarrow{I_j}) = \sum_{i=1}^{|C|} (\pi^{res}(C_i) - \pi^{j}(C_i))^2 \tag{10}$$

$$NSD(\overrightarrow{I_j}) = \begin{cases} 1 - \frac{Sd(\overrightarrow{I_j})}{2}, & if\ Sd(\overrightarrow{I_j}) \neq 2 \\ 1, & if\ Sd(\overrightarrow{I_j}) = 2 \end{cases} \tag{11}$$

where ANSCcc, ASCcc, ANSCms, and ASCmNs are abbreviations of Actual Non-smelly classes correctly classified (True Negative (TN)), Actual Smelly classes correctly classified (True Positive (TP)), Actual Non-smelly classes miss-classified as Smelly (False Negative (FN)), and Actual Smelly classes miss-classified as Non-smelly (False Positive (FP)), respectively. $Sd(\overrightarrow{I_j})$ (Similarity distance) (cf. Equation 10) is the distance between the resulting possibility distribution (π^{res}) and the real (initial) possibility distribution (π^{init}). This distance pertains to the interval [0, 2]. To give the $Sd(\overrightarrow{I_j})$ a significance similar to F-measure metric, we perform some modifications on $Sd(\overrightarrow{I_j})$ to attain the Normalized Similarity Distance ($NSD(\overrightarrow{I_j})$) (cf. Equation 11). This latter lies within a range of 0 and 1. According to the position of the most plausible class on both possibility distributions (π_{init} and π_{res}), the $NSD(\overrightarrow{I_j})$ value is added to one of these quantities: TP_dist (cf. Equation 6), TN_dist (cf. Equation 7), FP_dist (cf. Equation 8), or FN_dist (cf. Equation 9). After calculating these distances for all the unseen software classes, the $Precision\_dist$ (cf. Equation 4) and $Recall\_dist$ (cf. Equation 5) are measured to yield the *PF-measure_dist* (cf. Equation 3). In the following, an example is taken to explain the computation of *PF-measure_dist*. For instance, if π^{init}= [1 0.4] and π^{res} = [1 0.2] where the initial possibility distribution that corresponds to the software engineer opinion considers a given software class as Smelly and there is a possibility of 0.4 that it could be Non-smelly. Similarly, the predicted possibility distribution considers a given software class as Smelly, while there is a possibility of 0.2 that it could be Non-smelly. Thus, based on the measured distance between those possibility distributions, we will add the resulting distance to the TP_dist. Conversely, if π^{res}= [0.2 1], then the resulting distance will be joined to FP_dist. It is important to note that the working process

of *PF-measure_dist* is identical to the *F-measure* one in a certain environment. For more details please refer to Appendix D.

3.3 GA evolution operators

The employment of the GA to our problem requires the definition of the solution encoding. All along with a set of principal operators that are: (1) Population initialization, (2) Fitness assignment, (3) Mating selection, (4) Crossover, and (5) Mutation. The feasibility of the produced offsprings (PDTs) should be verified whenever the reproduction operators are triggered. Indeed, the fitness function is not presented here as it is already detailed in the previous section. The evaluation of a PDT requires its execution on the PBE and hence a 5-fold cross-validation strategy is performed to calculate the PDTs performance.

3.3.1 Individual encoding

To facilitate the PDT representation, we have decided to use a two-array encoding with breadth-first order. Figure 9 outlines this encoding for two cases: decision node and terminal node. For the case of the decision node, the first array incorporates the weights' vectors of the considered structural metrics such as: LOC, NOM, NOP, etc (cf. Appendix B) in each reached node. We notice that each decision node must contain the most discriminant feature (metric). As for the second array, it includes the splitting rule threshold. For the case of a terminal node, the first array has a similar structure as the case of a decision node but it finishes with a supplementary cell including a NULL value to mention that the actual node is a terminal one. However, the second array is the same as the case of an internal node but just with a possibility distribution π (e.g., [1 0.2]). For example, the terminal node number (3) contains a NULL value in the last cell of the first array. Such a choice is made to differentiate a terminal node from a decision node (Krętowski and Grześ 2005). In the second array, the last cell contains the possibility distribution (i.e., [1 0.6]), which represents the uncertainty aspect regarding the smelliness of a software class. The value [1 0.6] could be explained as follows: *"The passed software project class is fully possible smelly while the possibility that it could be non-smelly is 0.6"*. In a nutshell, assuming that a node is located at the breadth-first order in a position having index i, its left child is normally located at index $2i$, while the right one is located at the $(2i+1)$ position. It is important to know that metrics' thresholds are established according to an efficient discretization strategy (Krętowski and Grześ 2005), which is revealed in subsection 3.3.2. Among the important characteristics of our chosen encoding is that the feature (metric) selection is carried out in an implicit mode as the discarded metrics are allotted a zero as a weight. We notice that the considered quality metrics over this work are represented in Appendix B.

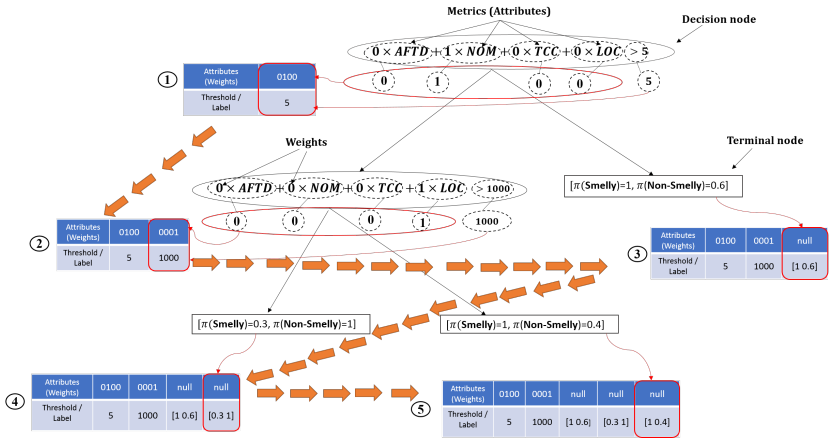


Fig. 9 Possibilistic decision tree and its corresponding structure. Decision nodes are represented by binary vectors in the first array cells and threshold values at the second array cells; while terminal nodes are represented by NULL values at the first array cells and possibility distributions at the second array cells.

3.3.2 Population initialization

In the initialization procedure, the GA initializes a set of N detectors (PDTs) according to three components, called (1) attribute selection, (2) partitioning strategy, and (3) stopping criterion and structure of leaves.

For *the attribute selection process*, each attribute is chosen randomly by assigning '1' as a weight to the chosen attribute (metric) while the remaining ones (the ignored metrics) are designated with '0' weights. Each decision node must contain only one discriminant feature (metric).

For *the partitioning strategy process*, after selecting an attribute at a given internal node, we apply a threshold discretization technique. However, one of the big problems in the current search-based detection approaches resides in threshold tuning. We note that the term threshold corresponds to the threshold of each structural metric existing in each decision node. It is important to know that the majority of the search-based smells detection researches did not specify the way to fix threshold value and solely claim that it is stochastically evolved throughout the optimization process. Unlikely, such a strategy could result in insignificant thresholds and hence degrades the effectiveness of the detectors' search process as well as their efficiency. To deal with this problem, we adopt in our ADIPE approach an existing discretization method, presented by Krętowski and Grześ (2005), for an effective definition of the threshold. This discretization method has already yielded interesting results when incorporated inside the basic decision tree induction algorithm, in particular the C4.5. The employed discretization technique has the following working principle: Once a structural metric (feature) is selected throughout a given node, its values are sorted in upward order. Afterward, boundary thresholds are identified according to the smelly class and non-smelly one as illustrated by Figure 10. As the class labels are represented by possibility distributions, we choose the most plausible classes (i.e., their possibility

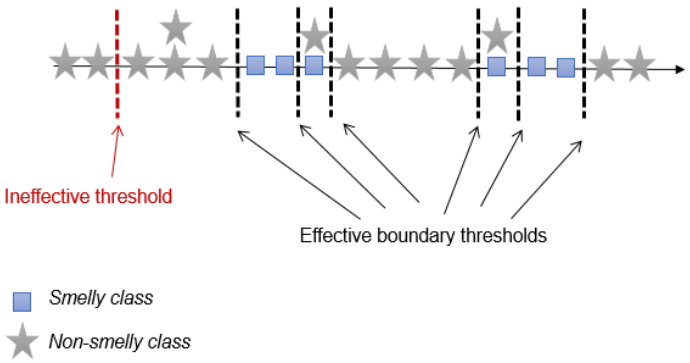


Fig. 10 The identification of the threshold boundaries for a randomly selected feature.

degrees equal to 1) to know if each software class is smelly or not. Then, among the obtained boundary thresholds for the given feature, one of them is chosen as a midpoint between a successive non-similar pair of examples, so that one example is Smelly and the other is Non-smelly. Finally, the threshold value of the given node is randomly picked among a set of effectively provided thresholds as demonstrated by Figure 10. This discretization strategy is performed whenever a new metric (feature) is selected inside a node.

Finally, the last component could be divided into two parts: (1) the structure of leaves and (2) the stopping criterion. For *the structure of leaves*, as our ADIPE approach is handling a PBE, where class labels are characterized with possibility distributions, the detectors (PDTs) are labeled by possibility distribution at each leaf node. It is important to note that deciding whether a particular node would be internal or leaf is an important decision for the initialization process. For this reason, we have tuned the parameter corresponding to this initialization using the Taguchi experimental design method. The results of this latter reveal that giving equal chances to both states ("internal node" and "leaf node") is a good choice. Therefore, each state has 50% of chances to be executed. Thus, the probability of each state occurrence is 0.5. Thus, the PBE located at the parent node of the given node must be partitioned according to the fixed threshold. Hence, the resulting possibility distributions' software classes should be combined to induce a representative possibility distribution denoted by π_{Rep} (Jenhani et al. 2008b). The π_{Rep} represents the different possibility degrees proportion of the different class labels values. Moreover, this possibility distribution is derived by the arithmetic mean of a set of possibility distributions where π_i (i=1..n) (Bouchon-Meunier et al. 1999), and it is calculated using equation 12. Then, π_{AM} should be normalized to obtain the π_{Rep}, which is given by equation 18. For the detection scenario as well as the identification one, the leave nodes contain possibility distributions. For instance, if a leaf node contains [1 0.2] this means that the detector fully supports the Class C_1 (i.e., the software class is Smelly) and less support the Class C_2 (i.e., Non-smelly) with possibility degree equals to 0.2. In the identification

task, the C_1 and C_2 (i.e., Smelly and Non-Smelly, respectively) are replaced with the smell type, i.e., C_1 refers to BLOB and C_2 corresponds to Non-BLOB.

$$\pi_{AM}(\omega_q) = \frac{1}{n} \times (\sum_{i=1}^{n} \pi_i(\omega_q)) \tag{12}$$

$$\pi_{Rep}(\omega_q) = \frac{\pi_{AM}(\omega_q)}{max_{q=1}^{|\Omega|}\{\pi_{AM}(\omega_q)\}} \tag{13}$$

In this work, the number of evaluations or the length of the tree (user-specified parameter) represents the stopping criterion of our approach.

3.3.3 Mating selection operators

As previously stated, one of the principal advantages of our ADIPE method is its capability to escape local optima and hence to converge to the globally-optimal detectors (PDTs). The principal mechanism that guarantees such behavior is the mating selection operator. Therefore, we have chosen the binary tournament selection operator (Brindle 1980) that can be described as follows:

- First, we pick $(N/2)$ parents for the reproduction, where N is the size of the population.
- Then, we run a loop of $(N/2)$ iterations. More precisely, at each iteration, two individuals (PDTs) are chosen randomly.
- Finally, the fittest parent will be placed throughout the mating pool.

Such strategic selection allows the selection of good and bad individuals to be chosen (i.e., we are managing the diversification of the population' individuals) with a biased preference for good individuals. Accordingly, the GA probabilistically accepts the degradation of fitness; which prevented to got stuck in a local optima and reach the global optima PDT(s).

3.3.4 Crossover and mutation operators

Once the individuals' selection process is done, we perform the one-point operator as it is shown in Figure 11. The working process is as follows. It begins by arbitrarily choosing a cut-point to locate a sub-tree in both parents. Then, the two selected sub-trees are swapped. Based on this fact, the values of the possibility distributions in each leaf node of those swapped sub-trees should be updated. It is worth noting that the update involves only the leaves nodes of the swapped sub-trees. The update of the possibility distribution of the leave nodes should be performed based on the new set of software classes located on the parent node of a given leaf node. For the mutation operator, only one type of change is possible, called weight change (cf. Figure 12). This operator changes the metrics' weights by allowing for the replacement of the current metric with another one. In other words, the current metric weight passes from '1' to '0' while the weight of a randomly chosen metric changes from '0' to '1'. For instance, according to the upper part of Figure 12, the LOC metric is changed to another metric,

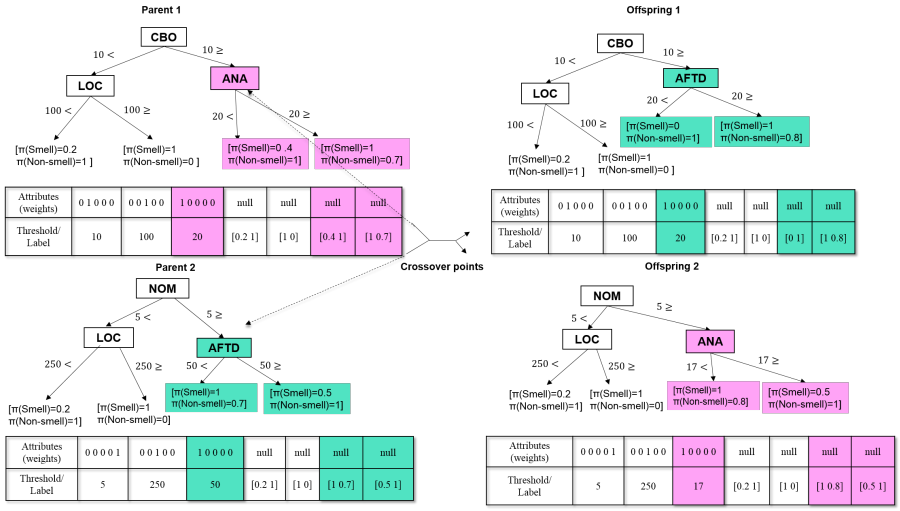


Fig. 11 One-point crossover operator.

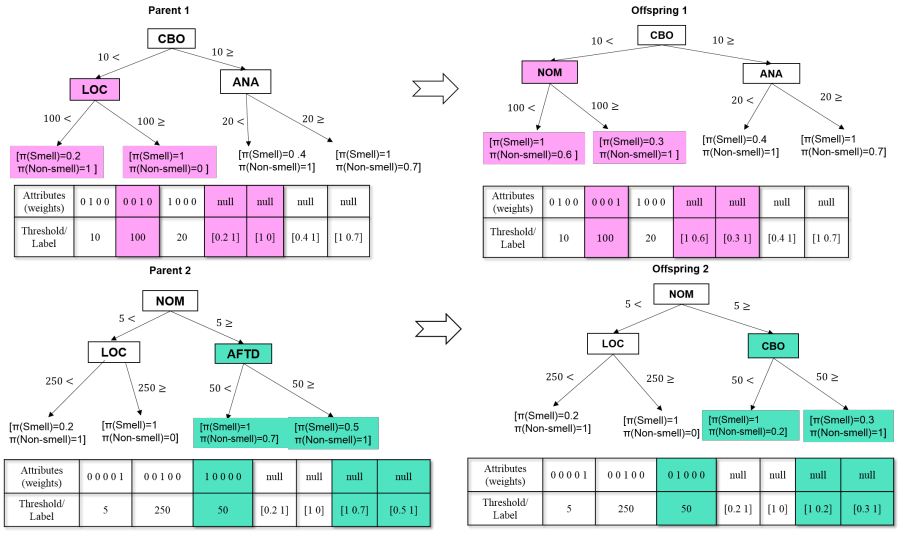


Fig. 12 Mutation operator.

which is NOM. Since the metric at the selected node is changed, we should update the possibility distributions of its leaves nodes. The update process of the possibility distribution for both crossover and mutation operations is performed using the π_{Rep} (cf. Equation 13). Moreover, the threshold tuning is performed randomly.

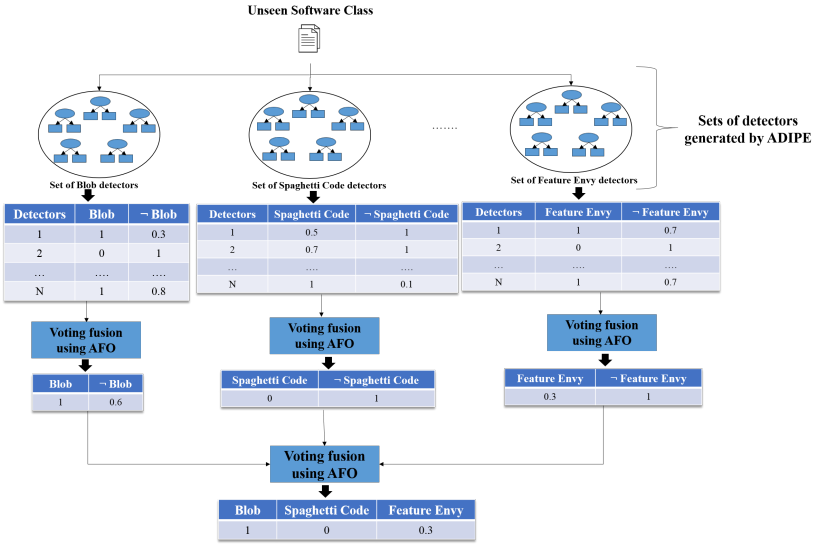


Fig. 13 Illustration of the use of PDTs generated by ADIPE for smell types identification on an unseen software class.

3.4 Possibilistic smell detectors application module

After the generation of a set of optimized detectors (PDTs) via the GA, the ADIPE tool is ready for use on unseen software classes to detect code smells and/or identify their types. In other words, the software engineer is allowed to employ some of the best detectors or all of them. As a result, a set of possibility distributions is obtained when applying the detectors on the unseen software classes. These latter will be aggregated by applying the AFO (cf. Equation 14) to produce a single possibility distribution that will support the software engineer when deciding about the smelliness of the unseen software class (Dubois and Prade 1994a). The AFO works as follows. For the case where the detectors are in disagreement, the AFO merges the decisions (i.e., possibility distributions) using the disjunctive operator (cf. Appendix C). Otherwise, the AFO merges the decisions based on the conjunctive operator (cf. Appendix C). We notice that in such a case, the detectors are in agreement. The choice of agreement or disagreement is related to the amount of conflict among the generated possibility distributions by the detectors. In other words, if $Inc(\pi_1 \wedge \pi_2) = 0$ means that the detectors are in agreement. Otherwise, the detectors are in disagreement (i.e., $Inc(\pi_1 \wedge \pi_2) \neq 0$). Figure 13 illustrates the use of PDTs generated by ADIPE for the identification of smell types on an unseen software class. Based on this figure, the output of identification (sets of PDTs for a specific smell type (i.e., a set for Blob, a set for Spaghetti Code, a set for Feature Envy, etc.)) are employed to identify the attributed smell type in a given unseen software class. In each set, the detectors will generate a number of decisions that will be aggregated using the AFO at the local level to obtain a single decision. This latter will be aggregated with the decisions of the other sets by employing the AFO aggregation method at the global level in order to obtain a single

decision that considers all the different existing smell types. It is important to note that we can obtain multiple fully possible smell types (i.e., multiple possibility degrees equal to 1). Finally, the generated decision will be provided to software engineers in order to have an idea about the existing smell types in the unseen class.

$$\forall \omega \in \Omega, \;\; \pi_{AD}(\omega) = max(\pi_{\wedge}(\omega),\, min(\pi_{\vee}(\omega),\, 1 - h(\pi_1, \pi_2))) \tag{14}$$

$$Inc(\pi_1 \wedge \pi_2)) = 1 - max(min(\pi_1, \pi_2)) \tag{15}$$

where $\pi_{\wedge}(\omega) = \frac{min(\pi_1(\omega), \pi_2(\omega))}{h(\pi_1, \pi_2)}$, $\pi_{\vee}(\omega) = max(\pi_1(\omega), \pi_2(\omega))$ and $h(\pi_1(\omega), \pi_2(\omega)) = 1 - Inc(\pi_1 \wedge \pi_2))$. The $h(\pi_1(\omega), \pi_2(\omega))$ represents the degree of agreement among two detectors (Dubois and Prade 1994b). Moreover, $\pi_{\wedge}$ and $\pi_{\vee}$ correspond to the conjunctive and disjunctive operators, respectively.

4 Evaluation of ADIPE performance

In this section, the performance of our ADIPE is evaluated with respect to the most prominent state-of-the-art existing works. In the following, we answer a number of research questions over a series of comparative experiments using six widely used software systems:

- **RQ1**: How does our ADIPE approach perform for the code smell detection problem in an uncertain environment? To answer this question, we report the performance of using adequate fitness function to deal with uncertain class labels as well as data imbalance problems. Moreover, we evaluate the effectiveness of ADIPE for the adaptation of adequate detectors (PDTs) in the aim to process the possibilistic class labels. Besides, we propose to use guided automated threshold tuning to achieve an optimized splitting. These are carried out over a set of comparative experiments regarding four state-of-the-art approaches.
- **RQ2**: How does our ADIPE approach perform to identify the code smell types? To answer this question, we report the performance of using a set of optimized detectors for each smell type to well identify the existing smell types. Thus, our ADIPE is compared with four respective state-of-the-art approaches.
- **RQ3**: How does our ADIPE approach perform to detect code smells and/or identify their types for uncertain class labels as well as data imbalance issues compared to a baseline PDT approach (e.g., NS-PDT)? It is important to know how far our proposed approach might surpass the baseline PDT with a greedy searching strategy.
- **RQ4**: What is the added value of outputting a possibility distribution instead of a single label for the software engineer? It is important to interpret the semantic meaning of each possibility value that is strictly less than 1. In brief, we show that such value allows defining the ranking of a particular software class (or a particular smell type) in terms of its prioritization for refactoring.

Table 2 Used Software in the experimentation. ECSS (Empirical Code Smell Studies) and DVDT (Designed by Various Developer Teams).

Systems	Version	NOC	KLOC	Description	ECSS	DVDT	Advantages
GanttProject	1.10.2	245	42	A platform for the scheduling of the projects	x	x	- Open-source - Code source publicly available - Rich in terms of code smells - Used in the empirical code smell studies - Designed by various developer teams
ArgoUML	0.19.8	200	284	A tool for UML modeling	x	x	
Xercess-J	2.7.0	991	240	Software for XML parsing	x	x	
JFreeChart	1.0.9	521	170	Java Library for the charts Java applications	x	x	
Apache Ant	1.7.0	1,839	327	Java library devoted to design Java applications	x	x	
Azureus	2.3.0.6	1,449	42	Peer to Peer (P2P) client program for sharing files	x	x	

Table 3 The number of existing smell types[10], the Number of Smelly Classes (Smelly Classes), the Number Non-Smelly Classes (Non-smelly Classes), and the Imbalance Ratio in the considered software systems within the BE

	Blob	DC	FE	LM	DuC	LPL	SC	FD	Classes	Smelly Classes	Non-smelly Classes	Imbalance Ratio
GanttProject	9	32	10	31	55	34	16	18	254	205	49	19.13%
ArgoUML	32	22	18	47	0	37	0	0	200	156	44	22%
Xercess-J	95	193	97	83	190	98	23	8	991	787	204	20.6%
JFreeChart	41	100	62	51	110	40	18	10	521	432	89	17.1%
Apache Ant	133	235	311	239	371	354	22	0	1839	1665	174	09.5%
Azureus	91	288	249	157	197	205	81	19	1449	1287	162	11.2%
Overall	401	870	747	608	923	768	160	55	5254	4532	722	13.7%

[10] The used smell types are Blob, Data Class (DC), Feature Envy (FE), Long Method (LM), Duplicate Code (DuC), Long Parameter List (LPL), Spaghetti Code (SC), Functional Decomposition (FD).

4.1 Subject Software

Our proposed tool is evaluated on a collection of widely used open-source OO applications that are: ArgoUML[3], Xerces-J[4], GanttProject[5], Apache Ant[6], Azureus[7], and JFreechart[8]. Table 2 sets out the key features of the software systems being considered in our experiment, where the columns (from the left side to the right one) present the system name, version, description, number of classes (NOC) as well as the number of lines of code (KLOC: thousands of code Lines), respectively. In our experimental study, we have considered eight smell types that are: Blob, Data Class (DC), Feature Envy (FE), Long Method (LM), Duplicate Code (DuC), Long Parameter List (LPL), Spaghetti Code (SC), and Functional Decomposition (FD) (cf. Appendix A). Table 3 presents the number of existing smell types, the Number of Classes, the Number of Smelly Classes, the Number Non-Smelly Classes, and the Imbalance Ratio in the considered software systems within the PBE. The constructed PBE is considered as our ground truth over this experimental study as it is an aggregation of the various simulated subjective uncertain opinions belonging to many experts (probabilistic classifiers). Indeed, these opinions are quantified to have the form of probability values.

[3] http://argouml.tigris.org/

[4] http://xerces.apache.org/xerces-j/

[5] https://sourceforge.net/projects/ganttproject/files/OldFiles/

[6] http://ant.apache.org

[7] http://vuze.com/

[8] http://www.jfree.org/jfreechart/

4.2 Selection of the baseline approaches

To compare ADIPE with state-of-the-art approaches, four relevant baselines were chosen: GP (Ouni et al. 2013), MOGP (Mansoor et al. 2017), BLOP (Sahin et al. 2014), and DECOR (Moha et al. 2010). The choice of these baseline approaches has been based on two important reasons. On the one side, the existing approaches (i.e., Mono-objective techniques (GP), Multi-objective techniques (MOGP), and the Bi-level ones (BLOP)) give us a broader overview of how our approach operates against existing approaches. On the other side, in order to evaluate the ADIPE performance against rule-based approaches, we have included the DECOR baseline as it represents the rule/heuristic based type. A concise summary of the working principle of each baseline approach is given as follows:

- GP: This approach generates a collection of IF-THEN rules via a metaheuristic method (i.e., genetic programming) that interacts with a BE containing code smells. Every solution is defined as a tree of rules for detection; where the inner nodes contain the quality metrics, as well as the class labels, are indicated by the leaf codes. These rules are developed by maximizing the number of detected software defects compared to those expected in the BE. As a result, GP obtained an average F-measure rate of 88% on six different software projects, when taking only three types of smells into account. We notice that the F-measure value is calculated based on the precision and recall values stated by (Ouni et al. 2013). Moreover, these authors have suggested in the same paper a multi-objective approach for refactoring the detected smells as much as possible using the NSGA-II method.
- MOGP: This approach is a multi-objective approach. From the standpoint of solution representation, MOGP is using the same GP encoding, while it modifies the fitness function. Also, MOGP has a similar framework as NSGA-II and it grows the trees through the optimization of two competing objectives. The first objective consists of maximizing the detection of existing code smells in the BE, while the second objective consists of reducing the detection of the well-design code pieces. To attain this aim, NSGA-II interacts with two BEs; The first one has smells, while the second one has well-designed codes. The authors explained the usage of well-designed codes under the reason that the employment of smells on their own would not permit the coverage of all smells. Therefore, a fragment of code may be regarded as a suspicious anti-pattern by maximizing the distance to a well-designed code example. Following the recorded precision and recall results, the mean F-measure for seven software projects is equal to 87% when five smell types are taken into account.
- BLOP: This approach has been proposed to address the lack of diversity problem that might have a BE. To this end, the authors have suggested modeling the problem of code smell detection as a bi-level optimization one as follows. In the upper-level, a collection of detection rules are evolved, while in the lower level, a collection of different artificial code smells are generated. From the fitness standpoint, the detection of real code smells as well as artificial ones are maximized by the upper level, while the probability of the event that the upper- level rules would detect code smells, is minimized by the lower level. Thus, the competition between the two

levels aims to: (1) generate rules characterized by important detection capability and (2) produce unknown artificial code smells for the diversification of the BE. According to the recorded precision and recall values, the average F-measure is around 90% for nine software projects, with respect to seven smell types.

– DECOR: This approach is a heuristic-based technique. DECOR employs a collection of rules, known as the "rule card", describing the intrinsic properties of a class contaminated by a smell. For instance, a Blob class is identified if the class contains: LCOM5 (Lack of Cohesion Of Methods) (Henderson-Sellers 1995) more than 20, a number of methods as well as attributes more than 20, a suffix in the collection {*"Process"*, *"Control"*, *"Command"*, *"Manage"*, *"Drive*, *"System"*} and an association of one-to-many with different data classes. The authors have shown the ability of DECOR to identify code smells with a mean F-measure of 80% (Moha et al. 2010).

4.3 Parameter configuration for ADIPE

The adjustment of algorithm parameters is an important aspect that is generally overlooked in metaheuristic search algorithms. It is important to realize that the setting of parameters greatly affects an algorithm's performance on a particular problem. Therefore, the ADIPE default parameters employed in the simulation part (cf. Table 4) are tuned by the trial-and-error technique (Eiben and Smit 2011), which is a common practice in the evolutionary computation field as well as the SBSE one (Karafotias et al. 2015; Boussaïd et al. 2017; Ramirez et al. 2018). In order to ensure a fair comparison between peer algorithms, we have employed the same stopping criterion for all the approaches under comparison. Hence, after 256,500 fitness evaluations, each run is halted. Such a choice is suitable for all the approaches under comparison including BLOP that uses two populations: (1) the upper level and (2) the lower one. Both levels evolve a population of 30 individuals. Indeed, the upper level population as well as the lower level one are evolved for 15 and 19 generations, respectively. These values are set in order to approximate the optimal lower level population, which is required to calculate the fitness function of its corresponding upper-level population. In this way, all algorithms including BLOP could meet the stopping criterion since with such settings the number of evaluations performed by BLOP is (30 x 15 x 30 x 19) = 256,500. In order to allow the reproducibility of ADIPE[11], we present the implementation details as follows. For the development of ADIPE, we have used the open source KEEL[12] (Knowledge Extraction based on Evolutionary Learning) platform (Alcalá-Fdez et al. 2011) to implement the GA algorithm with respect to our adopted solution encoding, fitness function, and variation operators. In ADIPE, a solution represents a PDT where each decision node is composed by two cells. The first cell contains a binary vector specifying the weights of the employed metrics, while the second cell includes the threshold value. For the case of a terminal node, the first cell contains a NULL value, while the second cell represents the possibility

[11] ADIPE is protected by a private Copyright

[12] http://www.keel.es

Table 4 The default parameters configuration.

Parameters	ADIPE	GP	MOGP	BLOP
Crossover rate	0.9	0.9	0.8	0.8
Mutation rate	0.1	0.5	0.2	0.5
Population size	200	100	100	30

distribution. For the crossover operator, we used the one-point crossover operator that starts by randomly choosing a cut-point to locate a sub-tree in the parent individuals. Then, it creates two child solutions by swapping the first sub-tree with the second one. After that, the weight change mutation operator is used to modify the weights of the metrics in order to allow the replacement of the current metric with another one. For the evaluation, for the evaluation of the generated solutions, we used as objective function the *PF-measure_dist* that evaluates the solutions based on the amount of closeness between the predicted software class labels comparing to their corresponding ones in the PBE[13]. Further details are given in Section 3.

4.4 Performance metrics

As we tackle an uncertain data classification issue, the performance metrics used should be able to quantify the performance of the different approaches considered in our study. The fact that the existing approaches ignore the uncertainty, this may negatively influence their performance. To alleviate this problem in our proposed approach, we have selected two appropriate measurements for the uncertain environment. The first one is *PF-measure_dist* that we have already introduced in section 3.2. The second one is the *IAC* (Information Affinity-based criterion) (cf. Equation 16) that was proposed by Jenhani et al. (2009). This metric uses the Affinity metric (cf. Equation 1) to measure the distance between the obtained possibility distribution ((π^{result})) and the original one (π^{init}). The *IAC* is used as a second metric since it takes into account only the uncertain class labels problem and does not give much importance to the imbalanced data issue, which is not the case with the *PF-measure_dist* metric that deals with both problems. If the *IAC* values are near to 1, this indicates that the generated detectors are more accurate as well as the produced possibility distributions are of high quality and faithful compared to the original ones. However, if the *IAC* values drop to 0, this implies that the obtained detectors are weak. It is important to note that in an uncertain environment the *PF-measure_dist* and *IAC* correspond to the *F-measure* and *PCC* (*Accuracy*), respectively (cf. Appendix D).

$$IAC = \frac{1}{n}\sum_{i=1}^{n} Aff(\pi_i^{init}, \pi_i^{ressult}) \tag{16}$$

[13] https://sites.google.com/view/sofienboutaib/accueil

4.5 Adopted statistical testing methodology

As GAs have stochastic behaviors, they generally yield different outputs from one run into another on the same software project (or problem). In such situations, it becomes difficult to compare stochastic smell detection approaches since the output may vary from one run to another. To deal with the stochastic nature of outcomes, researchers have suggested using statistical tests to detect the difference between the obtained results (Arcuri and Briand 2014). There are two possible types of tests: (1) Parametric tests requiring normalized data and (2) Non-parametric ones. To escape the data normality issue, we have chosen to use the Wilcoxon test (Conover and Conover 1980) by performing a pairwise comparison. Two hypothesis have been considered: H_0 implies that the two median values of the two compared algorithms are not significantly different over the number of runs and H_1 means the opposite. We use 5% as a significance rate, which means that the chance of rejecting H_0 is only 0.05. Besides the significance, it is important to quantify the difference between the results of the compared algorithms that's why the effect size should be reported. It is important to note that the Wilcoxon test allows only to verify if the obtained results are statically different or not. Nevertheless, such a test does not provide any indication concerning the difference magnitude. To achieve this goal, we adopted a non-parametric effect size measure called *Vargha-Delaney A test* (Vargha and Delaney 2000) to assess the improvement magnitude of the effect size. This latter could be: (1) "large" if A is less than 0.29 or greater than 0.71, (2) "medium" if A is less than 0.36 or greater than 0.64, or (3) "small" if A is less than 0.64 or greater than 0.64.

4.6 Analysis of the results

The following sub-section is dedicated to report and explain the obtained comparative results to tackle the three above-mentioned research questions as well as to demonstrate the effects of the key characteristics of the ADIPE approach. This latter include: (1) the possibilistic detectors for code smell detection (2) the GA to avoid falling into local optima, (3) the well-structure of smells detectors, and (4) the informed process of the threshold tuning. Besides, we demonstrated how ADIPE can be used for the detection task as well as the identification one.

4.6.1 Results for RQ1

To answer RQ1, we carry out a set of experiments on the six considered software applications with considering the uncertainty aspect. For ADIPE, we have transformed the original BE (with crisp classes) into a possibilistic one with possibility distributions across different class labels. The transformation process is performed to simulate the subjectivity of experts' opinions. In this work, we aim to prove the outperformance of our ADIPE approach for both cases. The first case is presented with an Uncertainty Level (UL) equals to 50%, which means that half of the BE class labels are covered by uncertainty. The second case refers to the crisp BE (i.e., UL=0 %), in which all the BE class labels are crisp. According to Table 5, ADIPE beats all the considered

state-of-the-art approaches in terms *PF-measure_dist*. The outperformance of ADIPE over the four remaining approaches (i.e., DECOR, GP, MOGP, and BLOP) could be explained by the fact that our ADIPE approach considers the uncertainty aspect over the process of the solution evaluation, while the competitors approaches ignore this aspect. We notice that the adopted fitness function by the ADIPE (*PF-measure_dist*) had proven its efficacy in dealing with the uncertain class labels problem since it is insensitive to the data imbalance problem. DECOR obtained the worst results in terms of *PF-measure_dist*. The DECOR results could be justified by the fact that its rules are designed manually and without considering the uncertainty factor. As for the *IAC* metric results, they are almost similar to those presented by the *PF-measure_dist* since a certain case represents a sub-case of the uncertain one. From the Precision and Recall viewpoints, ADIPE has shown its outperformance against its competitors where its obtained values belongs to [0.8724, 0.9345] for the Precision and [0.9420, 0.8809] for the Recall. However, BLOP obtained the second best performance with 0.4416 and 0.4837 for the Precision and the Recall measures, respectively. This could be explained by the fact that our ADIPE approach has a fitness function (i.e., *PF-measure_dist*) that takes into consideration the uncertainty, which is not the case of the remaining approaches (i.e., DECOR, GP, MOGP, and BLOP). Moreover, the calculation of the adopted fitness function is based on distances, these latter allow translating the hyper-planes in the data space with the aim to well-distinguish minority instances (e.g., non-smelly classes) from the majority ones (e.g., smelly classes). Therefore, when a data point is located at the boundaries of the minority class (non-smelly instances), the X value would express the non-possibility of belonging to the majority class (smelly instances). By using the information provided by X, the induction (classifier construction) algorithm would be better guided to discover more effective splitting hyper-planes. Table 5 shows the obtained A statistic results of the five algorithms under comparison. One can observe that ADIPE succeeds to obtain an A value greater than 0.9 (large) based on the *PF-measure_dist* and the *IAC* metrics on all the considered software projects. It is important to know that a certain case corresponds to the ground truth. In other words, the possibility distribution for a certain case could be represented by a binary vector including only a value of 1 (i.e., the real class label) while the remaining values are set to 0. This demonstrates that ADIPE is able to deal with uncertain environment. Figure 14 displays the Boxplots of the compared approaches for the detection case under an uncertain environment. This Figure clearly demonstrates the superiority of ADIPE over its competitors in terms *PF-measure_dist* and *AUC*. The obtained box plots are compliant with the reported results in Table 5.

Table 6 presents the results of the used metrics for the five detection approaches in the case of a certain environment, i.e., ULevel = 0%. For the case of the crisp (certain) environment, the class labels are certain as their BE. Hence, the performance of the *PF-measure_dist* acts similarly to the F-measure one. Based on this table, our ADIPE approach performs better than the other four considered approaches in terms of *PF-measure_dist*. The surpass of ADIPE across its competitors could be explained as follows. On the one hand, the adopted fitness functions by the considered approaches (excluding ours) are not adequate to the data imbalance that could trick the search process of the generated detection rules. In contrast, the fitness function adopted by ADIPE is insensitive to the problem of imbalanced data as the *PF-measure_dist*

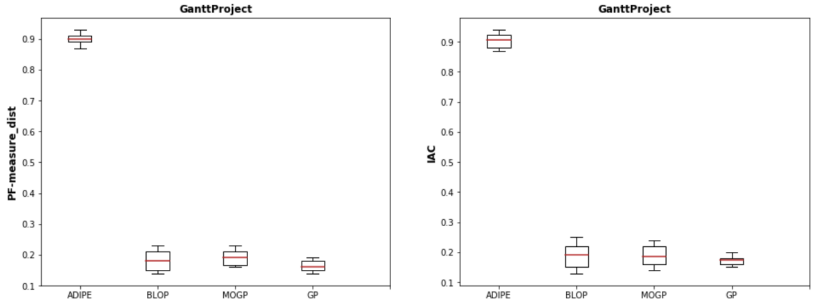


Fig. 14 Boxplots of the *PF-measure_dist* and *IAC* values (i.e., an uncertain environment) for the detection case on the GanttProject.

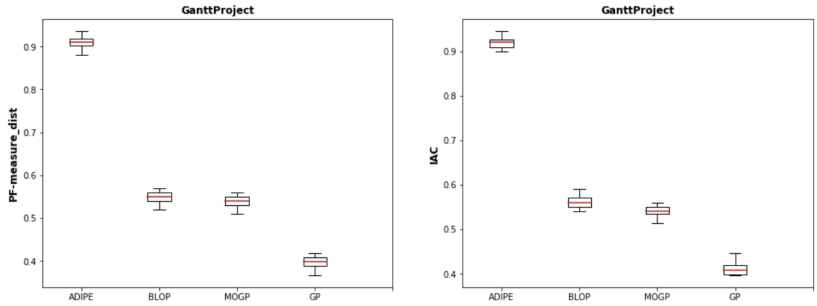


Fig. 15 Boxplots of the *PF-measure_dist* and *IAC* values (i.e., a certain environment) for the detection case on the GanttProject. The *PF-measure_dist* and the *IAC* correspond to the *F-measure* and *PCC* in the certain environment.

behaves similarly to the F-measure especially under certain environments. On the other hand, the considered methods (excluding DECOR) randomly determine the thresholds. Table 6 shows that the obtained Recall and Precision values of ADIPE surpass those of the remaining approaches, which could be explained by the fact that ADIPE is well suitable for the imbalanced environment while the remaining approaches are unable to deal with imbalanced data. Thus, ineffective slicing values may be obtained. Table 6 shows the obtained A statistic results of ADIPE, DECOR, GP, MOGP, and BLOP using the *PF-measure_dist* and the *IAC* metrics. One can see that ADIPE succeeds to obtain an A value higher than 0.86 (large) on all the considered software projects. The obtained experimental results are confirmed by the box plots of Figure 15. The analysis of these box plots demonstrates the high performance of ADIPE based on the *PF-measure_dist* and *AUC* for the detection in a certain environment.

In nutshell, the ADIPE performance surpass could be explained as follows. For the uncertain environment viewpoint, the *PF-measure_dist* is a good metric to cope with the uncertainty located at the BE class labels., while for the certain environment viewpoint, the *PF-measure_dist* mimics the behavior of the F-measure since this latter is insensitive towards the problem of imbalanced data.

Table 5 *PF-measure_dist*, *IAC*, *Precision*, and *Recall* median scores of ADIPE, DECOR, GP, MOGP, and BLOP for 31 runs of the detection task at the uncertainty level UL=50%.

	ADIPE				DECOR				GP				MOGP				BLOP			
Projects	PF_d	IAC	Precision	Recall	PF_d	IAC	Precision	Recall	PF_d	IAC	Precision	Recall	PF_d	IAC	Precision	Recall	PF_d	IAC	Precision	Recall
GanttProject	**0.9104**	**0.9288**	**0.9016**	**0.9173**	0.1519	0.1721	0.3604	0.3717	0.1753	0.1908	0.4187	0.4312	0.2185	0.2243	0.5680	0.5732				
	(+ + + +)	(+ + + +)	(+ + + +)	(+ + + +)	(- + +)	(+ + +)	(- + +)	(+ + +)	(+ +)	(- +)	(+ +)	(+ +)	(-)	(-)	(-)	(-)	0.2252	0.2436	0.5788	0.5902
	(l l l)	**(l l l)**	**(l l l)**	**(l l l)**	(s m m)	(m m m)	(s m m)	(m m m)	(m m)	(s m)	(m m)	(m m)	(s)	(s)	(s)	(s)				
ArgoUML	**0.9187**	**0.9452**	**0.9004**	**0.9120**	0.128	0.1428	0.3619	0.3621	0.1522	0.1745	0.3753	0.3810	0.204	0.2125	0.5662	0.5839				
	(+ + + +)	(+ + + +)	(+ + + +)	(+ + + +)	(- + +)	(- - +)	(- + +)	(+ + +)	(+ +)	(- +)	(+ +)	(+ +)	(-)	(-)	(-)	(-)	0.2249	0.2417	0.5896	0.5938
	(l l l)	**(l l l)**	**(l l l)**	**(l l l)**	(s m m)	(s m m)	(s m m)	(m m m)	(m m)	(s m)	(l l)	(l l)	(s)	(s)	(s)	(s)				
Xercess-J	**0.9047**	**0.9294**	**0.8985**	**0.9061**	0.0814	0.0917	0.2060	0.2110	0.1413	0.1609	0.3341	0.3591	0.1943	0.2005	0.4819	0.4916				
	(+ + + +)	(+ + + +)	(+ + + +)	(+ + + +)	(+ + +)	(+ + +)	(+ + +)	(+ + +)	(+ +)	(+ +)	(+ +)	(+ +)	(-)	(-)	(-)	(-)	0.2078	0.2160	0.5204	0.5541
	(l l l)	**(l l l)**	**(l l l)**	**(l l l)**	(m m l)	(m m l)	(m m l)	(m m l)	(m m)	(m m)	(l l)	(l l)	(s)	(s)	(s)	(s)				
JFreeChart	**0.9285**	**0.9557**	**0.9174**	**0.9256**	0.0807	0.0906	0.1792	0.1952	0.1251	0.1380	0.3119	0.3406	0.1734	0.176	0.4126	0.4276				
	(+ + + +)	(+ + + +)	(+ + + +)	(+ + + +)	(- + +)	(- + +)	(+ + +)	(+ + +)	(+ +)	(+ +)	(+ +)	(+ +)	(-)	(-)	(+)	(+)	0.1805	0.1913	0.5120	0.5279
	(l l l)	**(l l l)**	**(l l l)**	**(l l l)**	(s m l)	(s m l)	(l l l)	(l l l)	(m m)	(m m)	(l l)	(l l)	(s)	(s)	(m)	(m)				
Azureus	**0.9264**	**0.9408**	**0.9163**	**0.9283**	0.0495	0.0653	0.1332	0.1473	0.1058	0.1211	0.2614	0.2761	0.1499	0.1595	0.3625	0.3721				
	(+ + + +)	(+ + + +)	(+ + + +)	(+ + + +)	(+ + +)	(- + +)	(+ + +)	(+ + +)	(+ +)	(+ +)	(+ +)	(+ +)	(-)	(-)	(+)	(+)	0.1644	0.1682	0.4316	0.4421
	(l l l)	**(l l l)**	**(l l l)**	**(l l l)**	(m m m)	(s m m)	(l l l)	(l l l)	(m m)	(m m)	(l l)	(l l)	(s)	(s)	(m)	(m)				
Apache Ant	**0.9165**	**0.9372**	**0.9210**	**0.9331**	0.0451	0.0624	0.1223	0.1315	0.0827	0.0988	0.1079	0.1121	0.1494	0.1512	0.3312	0.3452				
	(+ + + +)	(+ + + +)	(+ + + +)	(+ + + +)	(+ + +)	(+ + +)	(+ + +)	(+ + +)	(+ +)	(+ +)	(+ +)	(+ +)	(-)	(-)	(+)	(+)	0.1540	0.1637	0.4014	0.4106
	(l l l)	**(l l l)**	**(l l l)**	**(l l l)**	(m m m)	(m m m)	(l l l)	(l l l)	(m m)	(m m)	(l l)	(l l)	(s)	(s)	(m)	(m)				

- The sign "+" at the i^{th} position means that the algorithm metric median value (*PF-measure_dist* (abbreviated by PF_d), *IAC*, *Precision*, or *Recall* median value) is statically different from the i^{th} algorithm value. The sign "-" means the opposite.
- The effect sizes values (small (s), medium (m), and large (l)) using the A-statistics are given.
- Best obtained metrics values are highlighted in Bold. Second-best obtained metrics values are underlined.

Table 6 *PF-measure_dist*, *IAC*, *Precision*, and *Recall* median scores of ADIPE, DECOR, GP, MOGP, and BLOP for 31 runs of the detection task at the uncertainty level UL=0%. [15]

Projects	ADIPE				DÉCOR				GP				MOGP				BLOP			
	PF_d	IAC	Precision	Recall	PF_d	IAC	Precision	Recall	PF_d	IAC	Precision	Recall	PF_d	IAC	Precision	Recall	PF_d	IAC	Precision	Recall
GanttProject	**0.9177** (+ + + +) **(l l l)**	**0.939** (+ + + +) **(l l l)**	**0.9241** (+ + + +) **(l l l)**	**0.9318** (+ + + +) **(l l l)**	0.3534 (- + +) (s m m)	0.3781 (+ + +) (m m m)	0.3623 (- + +) (s m m)	0.3729 (+ + +) (m m m)	0.4056 (+ +) (m m)	0.4216 (- +) (s m)	0.4206 (+ +) (m m)	0.4393 (+ +) (m m)	0.554 (-) (s)	0.5574 (-) (s)	0.5730 (-) (s)	0.5810 (-) (s)	0.5580	0.5691	0.5921	0.6152
ArgoUML	**0.9327** (+ + + +) **(l l l)**	**0.9604** (+ + + +) **(l l l)**	**0.9306** (+ + + +) **(l l l)**	**0.9417** (+ + + +) **(l l l)**	0.3586 (- + +) (s m m)	0.33428 (- - +) (s s m)	0.3645 (- + +) (s m m)	0.3683 (+ + +) (m m m)	0.3604 (+ +) (m m)	0.3846 (- +) (s m)	0.3812 (+ +) (l l)	0.3925 (+ +) (l l)	0.5528 (-) (s)	0.5481 (-) (s)	0.5731 (-) (s)	0.5951 (-) (s)	0.5778	0.5659	0.5918	0.6025
Xercess-J	**0.9208** (+ + + +) **(l l l)**	**0.9437** (+ + + +) **(l l l)**	**0.9176** (+ + + +) **(l l l)**	**0.9233** (+ + + +) **(l l l)**	0.1971 (+ + +) (m m l)	0.22118 (+ + +) (m m l)	0.2087 (+ + +) (m m l)	0.2124 (+ + +) (m m l)	0.3204 (+ +) (m m)	0.3375 (- -) (s s)	0.3416 (+ +) (l l)	0.3711 (+ +) (l l)	0.4709 (-) (s)	0.4713 (-) (s)	0.4883 (-) (s)	0.4916 (-) (s)	0.5142	0.5211	0.5319	0.5541
JFreeChart	**0.9466** (+ + + +) **(l l l)**	**0.9617** (+ + + +) **(l l l)**	**0.9523** (+ + + +) **(l l l)**	**0.9637** (+ + + +) **(l l l)**	0.1708 (- + +) (s m l)	0.20368 (- + +) (s m l)	0.1807 (+ + +) (l l l)	0.1983 (+ + +) (l l l)	0.3063 (+ +) (m m)	0.3096 (+ +) (m m)	0.3179 (+ +) (l l)	0.3452 (+ +) (l l)	0.4006 (-) (s)	0.4356 (-) (s)	0.4325 (+) (l)	0.4625 (+) (l)	0.5003	0.5098	0.5346	0.5519
Azureus	**0.9389** (+ + + +) **(l l l)**	**0.9596** (+ + + +) **(l l l)**	**0.9412** (+ + + +) **(l l l)**	**0.9507** (+ + + +) **(l l l)**	0.1297 (+ + +) (m m m)	0.14618 (- + +) (s m m)	0.1379 (+ + +) (l l l)	0.1491 (+ + +) (l l l)	0.2415 (+ +) (m m)	0.2728 (- +) (s m)	0.2706 (+ +) (l l)	0.2819 (+ +) (l l)	0.3560 (-) (s)	0.3689 (-) (s)	0.3809 (+) (m)	0.3958 (+) (m)	0.4233	0.437	0.4478	0.4536
Apache Ant	**0.9447** (+ + + +) **(l l l)**	**0.9577** (+ + + +) **(l l l)**	**0.9508** (+ + + +) **(l l l)**	**0.9641** (+ + + +) **(l l l)**	0.1120 (+ + +) (m m m)	0.12168 (+ + +) (s m m)	0.1296 (+ + +) (l l l)	0.1362 (+ + +) (l l l)	0.2253 (+ +) (m m)	0.2255 (- +) (s m)	0.1083 (+ +) (l l)	0.1176 (+ +) (l l)	0.3201 (-) (s)	0.3336 (-) (s)	0.3011 (-) (s)	0.3208 (-) (s)	0.3210	0.3527	0.3572	0.3619

- The sign "+" at the i^{th} position means that the algorithm metric median value (*PF-measure_dist* (abbreviated by PF_d), *IAC*, *Precision*, or *Recall* median value) is statically different from the i^{th} algorithm value. The sign "-" means the opposite.
- The effect sizes values (small (s), medium (m), and large (l)) using the A-statistics are given.
- Best obtained metrics values are highlighted in Bold. Second-best obtained metrics values are underlined.

[15] In the certain environment, the *PF-measure_dist* (abbreviated by PF_d) and *IAC* measures become *F*-measure (abbreviated by Fm) and *PCC* measures, respectively.

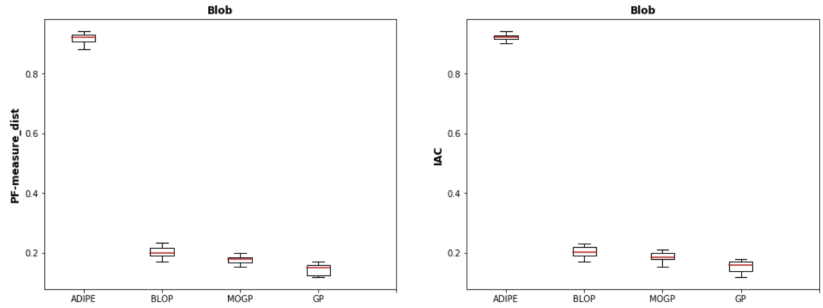


Fig. 16 Boxplots of the *PF-measure_dist* and *IAC* values (i.e., an uncertain environment) for the identification of the Blob smell type.

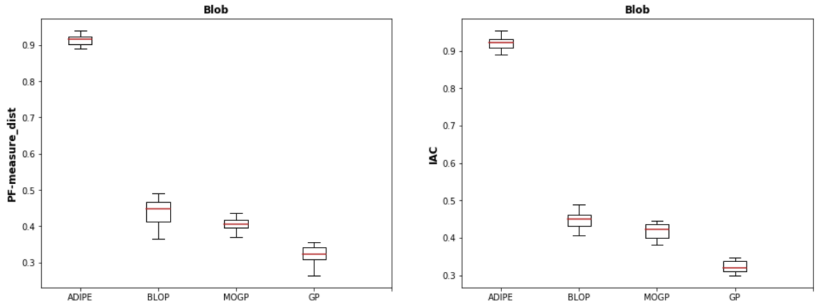


Fig. 17 Boxplots of the *PF-measure_dist* and *IAC* values (i.e., a certain environment) for the identification of the Blob smell type. The *PF-measure_dist* and the *IAC* correspond to the *F-measure* and *PCC* in the certain environment.

4.6.2 Results for RQ2

To answer RQ2, we aim over this subsection to evaluate the performance of the compared approaches on the identification of the smell type problem for both environments: (1) the uncertain environment (having uncertain class labels) and (2) the certain one (having only crisp class labels). It is important to know that the identification process is harder than the detection one as its imbalance ratio is greater than that of the detection process. For the realization of the comparative experiments, for every smell type, we combined all the software systems into only one BE and then, we determine the minority class. However, in this task, we noted that the number of classes that are not smelly are larger than the number of the smelly ones, which causes a significant imbalance ratio throughout the BE. For the uncertain environment, every software has a class label in the form of a possibility distribution, which is represented by a vector of two real numbers. Each value of the vector is a possibility degree which indicates the membership degree of a given software class to each of the following class labels: (1) Smell and (2) Non-Smelly. Table 7 reports the *PF-measure_dist* and *IAC* values of ADIPE, DECOR, GP, MOGP, and BLOP. One can see from this table that ADIPE outperforms its competitors since the identification process is performed in an uncertain environment. Based on the same table, ADIPE

has shown its outperformance against its competitors through the Precision and Recall values. This could be explained by the fact that the PDTs detectors used by ADIPE are able to overcome the smell disjuncts and data overlap problems encountered when dealing with imbalanced data. Table 7 shows the obtained A statistic results of ADIPE, DECOR, GP, MOGP, and BLOP using the *PF-measure_dist* and the *IAC* metrics for the smell types identification case under an uncertain environment. The obtained results reveal that ADIPE succeeds to obtain a value greater than 0.91 (large) over the eight considered smell types. Thus, we can conclude that our approach significantly exceeds its competitors. Figure 16 shows the boxplots obtained by ADIPE, GP, MOGP, and BLOP for the identification of the Blob smell type using the*PF-measure_dist* and the *AUC*. This Figure illustrates the outperformance of ADIPE against its competitors in identifying the Blob smell type. Therefore, it does not contradict the obtained results of Table 7.

For the case of a certain environment, all the BEs class labels are certain. According to Table 8, the ADIPE approach outperforms all the remaining approaches in terms of *PF-measure_dist*. Similar results are obtained for the *IAC* metric. Moreover, the obtained Recall and Precision values of ADIPE surpass those of the remaining approaches. These results are explained by the fact that the adopted fitness function allows us to obtain detectors suitable for the case of imbalanced data while the remaining approaches have obtained bad detectors since their fitness functions are not adequate to cope with the data imbalance problem. These results are explained by the fact that the adopted fitness function allows us to obtain detectors suitable for the case of imbalanced data while the remaining approaches have obtained bad detectors since their fitness functions are not adequate to cope with the data imbalance problem. In summary, the deterioration in performance did not influence all the approaches of the same size. The results quality demonstrated by ADIPE is somewhat lower compared to its performance quality over the detection process. However, BLOP, MOGP, DECOR, GP performance indicators' values are significantly reduced. These results can be explained as follows. For the case of the DECOR approach, the obtained results were extremely poor since its rules are predefined, which will make it not adequate to the data imbalance and the uncertain class labels problems. In contrast, for the case of GP, MOGP, and BLOP, their results are varying between poor and very poor because their identification processes are evolved using meta-heuristic algorithms. To this end, they can detect some smells by chance. Table 8 shows the obtained A statistic results of the five algorithms under comparison using the *PF-measure_dist* and the *IAC* metrics for the identification case under a certain environment. The obtained results shows that ADIPE succeeds to obtain a value greater than 0.8 (large) over all the considered smell types. Therefore, we can deduce that ADIPE significantly surpasses DECOR, GP, MOGP, and BLOP. Figure 17 displays the box plots obtained by the four search-based approaches in terms *PF-measure_dist* and *AUC* in a certain environment. This figure clearly shows the high performance of ADIPE with regard to the Blob smell type.

Generally speaking, we can conclude from the different obtained results that the deterioration in performance did not influence all the approaches of the same size. Our ADIPE succeeds to obtain better results in the detection and the identification processes. However, BLOP, MOGP, DECOR, GP have shown their deeper weaknesses in dealing with both processes. For the case of the DECOR approach, its results were

extremely poor as its rules are predefined. In contrast, for the case of GP, MOGP, and BLOP, their results are varying between poor and very poor because their identification processes are evolved using meta-heuristic algorithms. To this end, they can detect some smells by chance.

4.6.3 Results for RQ3

To answer RQ3, we recall that the main objective of this research question is to compare the performance of our ADIPE against a baseline PDT. For the comparison, we used NS-PDT as it keeps the uncertainty in the DT (Decision Tree) throughout the building process while the other PDT methods (i.e., SIM-PDT (Jenhani et al. 2008a) and Clust-PDT (Jenhani et al. 2009)) get rid of uncertainty. In general, the PDT approach is based on a greedy search method to construct their classifiers. In other words, the greedy search is performed to select an attribute at each node of the PDT. Also, the threshold values are fixed by doing a greedy search. We compare the two approaches in an uncertain environment and a crisp one for both tasks: (1) detection and (2) identification.

For the detection of code smells under uncertainty (cf. Table 9), ADIPE succeeds to obtain the best performance in terms of *PF-measure_dist* and *IAC*.

For the identification (cf. Table 10), ADIPE outperforms the baseline approach in the identification of eight smell types in both environments (i.e., certain and uncertain). These results could be explained by the following reasons. On the one hand, the ADIPE approach can avoid falling into the local optima as well as approaching from the global optima thanks to the used GA. However, the baseline approach performs a greedy search over the search space which will make it got stuck into a local optima. In the same context, the method employed by ADIPE allows defining effective and meaningful thresholds in contrast to the baseline PDT' threshold that is carried out by greedy search. On the other hand, the ADIPE adopted a well-adequate fitness function able to manage the uncertainty problem as well as the data imbalanced one, while the PDT can manage only the uncertainty. That's why it has a medium quality of results in the detection process that is characterized with lower data imbalance ratio. Based on these facts, ADIPE generates a set of optimized detectors for the detection task and also a set of optimized detectors for each smell type. However, the baseline PDT produces only non-optimized PDTs. It is important to notice that the degradation of the results of ADIPE in the identification task could be explained by the fact that the imbalance ratio is extremely higher. The results degradation of the baseline approach is remarkable. Tables 9 and 10 show the obtained A statistic results of ADIPE and baseline PDT in the detection and identification cases under the certain and uncertain environments. The obtained results clearly demonstrate that ADIPE significantly exceeds the baseline PDT since its A statistic values range between [0.75, 0.84] (large).

4.6.4 Results for RQ4

To answer RQ4, we need to discuss the added value of the outputted possibility distribution. Traditional detectors, including SBSE ones, return directly the smelliness result for each software class as a label (1: Smelly and 0: Not Smelly). Differently,

Table 7 *PF-measure_dist*, *IAC*, *Precision*, and *Recall* median scores of ADIPE, DECOR, GP, MOGP, and BLOP for 31 runs of the identification task at the uncertainty level L=50%.

Code Smells	ADIPE				DECOR				GP				MOGP				BLOP			
	PF_d	IAC	Precision	Recall	PF_d	IAC	Precision	Recall	PF_d	IAC	Precision	Recall	PF_d	IAC	Precision	Recall	PF_d	IAC	Precision	Recall
Blob	**0.9228** (+ + + +) **(l l l)**	**0.933** (+ + + +) **(l l l)**	**0.9345** (+ + + +) **(l l l)**	**0.9420** (+ + + +) **(l l l)**	0.0853 (- + +) (s m m)	0.0927 (- + +) (s m m)	0.3545 (- + +) (s m m)	0.3751 (- + +) (s m m)	0.1499 (+ +) (m m)	0.1615 (+ +) (m m)	0.348 (+ +) (l l)	0.3589 (+ +) (l l)	0.1826 (-) (s)	0.1925 (-) (s)	0.4281 (-) (s)	0.4305 (-) (s)	0.2086	0.2190	0.4416	0.4837
Data Class	**0.8941** (+ + +)(l l l)	**0.9207** (+ + +) **(l l l)**	**0.9080** (+ + +) **(l l l)**	**0.9141** (+ + +) **(l l l)**	N/A	N/A	N/A	N/A	0.1210 (+ +) (m m)	0.1243 (+ +) (m m)	0.2746 (+ +) (l l)	0.2820 (+ +) (l l)	0.1412 (-) (s)	0.1453 (-) (s)	0.3357 (-) (s)	0.3486 (-) (s)	0.1839	0.1906	0.3851	0.3941
Feature Envy	**0.8861** (+ + +) **(l l l)**	**0.9037** (+ + + +) **(l l l)**	**0.8913** (+ + +) **(l l l)**	**0.9065** (+ + +) **(l l l)**	N/A	N/A	N/A	N/A	0.1137 (+ +) (m m)	0.1195 (+ +) (m m)	0.2674 (+ +) (l l)	0.2712 (+ +) (l l)	0.1369 (-) (s)	0.1451 (-) (s)	0.3412 (-) (s)	0.3519 (-) (s)	0.1785	0.1821	0.3717	0.3846
Long Method	**0.8794** (+ + +) **(l l l)**	**0.896** (+ + +) **(l l l)**	**0.8831** (+ + +) **(l l l)**	**0.9016** (+ + +) **(l l l)**	N/A	N/A	N/A	N/A	0.1079 (+ +) (m m)	0.1107 (+ +) (m m)	0.2480 (+ +) (l l)	0.2605 (+ +) (l l)	0.1285 (-) (s)	0.1363 (-) (s)	0.3089 (-) (s)	0.3146 (-) (s)	0.1502	0.1528	0.3456	0.3517
Duplicate Code	**0.8623** (+ + +) **(l l l)**	**0.8849** (+ + +) **(l l l)**	**0.8736** (+ + +) **(l l l)**	**0.8919** (+ + +) **(l l l)**	N/A	N/A	N/A	N/A	0.0834 (+ +) (m l)	0.0890 (+ +) (m l)	0.2189 (+ +) (l l)	0.2276 (+ +) (l l)	0.1067 (+) (m)	0.1124 (+) (m)	0.3076 (-) (s)	0.3305 (-) (s)	0.1279	0.1348	0.3374	0.3528
Long Parameter List	**0.8721** (+ + +) **(l l l)**	**0.897** (+ + +) **(l l l)**	**0.8994** (+ + +) **(l l l)**	**0.9116** (+ + +) **(l l l)**	N/A	N/A	N/A	N/A	0.0663 (+ +) (l l)	0.0724 (+ +) (l l)	0.1522 (+ +) (l l)	0.1619 (+ +) (l l)	0.0982 (-) (s)	0.1072 (-) (s)	0.2429 (+) (m)	0.2466 (+) (m)	0.1137	0.1201	0.3014	0.3252
Spaghetti Code	**0.8631** (+ + + +) **(l l l)**	**0.887** (+ + + +) **(l l l)**	**0.8810** (+ + + +) **(l l l)**	**0.8952** (+ + + +) **(l l l)**	0.0350 (+ + +) (l l l)	0.0447 (+ + +) (l l l)	0.1139 (+ + +) (l l l)	0.1308 (+ + +) (l l l)	0.0495 (+ +) (m l)	0.0625 (+ +) (m l)	0.1130 (+ +) (l l)	0.1222 (+ +) (l l)	0.0797 (-) (s)	0.0851 (-) (s)	0.1987 (+) (m)	0.2003 (+) (m)	0.0927	0.1030	0.2617	0.2789
Functional Decomposition	**0.8545** (+ + + +) **(l l l)**	**0.8751** (+ + + +) **(l l l)**	**0.8724** (+ + + +) **(l l l)**	**0.8809** (+ + + +) **(l l l)**	0.0226 (+ + +) (l l l)	0.0335 (+ + +) (l l l)	0.0810 (+ + +) (l l l)	0.1086 (+ + +) (l l l)	0.0476 (+ +) (m l)	0.0584 (- +) (s l)	0.1107 (+ +) (l l)	0.1198 (+ +) (l l)	0.0529 (-) (s)	0.0615 (-) (s)	0.1611 (+) (m)	0.1681 (+) (m)	0.0603	0.0716	0.2145	0.2239

- The sign "+" at the i^{th} position means that the algorithm metric median value (*PF-measure_dist* (abbreviated by PF_d), *IAC*, *Precision*, or *Recall* median value) is statically different from the i^{th} algorithm value. The sign "-" means the opposite.
- The effect sizes values (small (s), medium (m), and large (l)) using the A-statistics are given.
-The N/A signifies that the given approach is Not Applicable (N/A) on the corresponding smell.
- Best obtained metrics values are highlighted in Bold. Second-best obtained metrics values are underlined.

Table 8 *PF-measure_dist*, *IAC*, *Precision*, and *Recall* median scores of ADIPE, DECOR, GP, MOGP, and BLOP for 31 runs of the identification task at the uncertainty level L=0%.[17].

Code Smells	ADIPE				DECOR				GP				MOGP				BLOP			
	PF_d	IAC	Precision	Recall	PF_d	IAC	Precision	Recall	PF_d	IAC	Precision	Recall	PF_d	IAC	Precision	Recall	PF_d	IAC	Precision	Recall
Blob	**0.9228** (+ + + +) **(l l l)**	**0.9483** (+ + + +) **(l l l)**	**0.9407** (+ + + +) **(l l l)**	**0.9511** (+ + + +) **(l l l)**	0.346 (- + +) (s m m)	0.3520 (- + +) (s m m)	0.3729 (- + +) (s m m)	0.3819 (- + +) (s m m)	0.3316 (+ +) (m m)	0.3484 (+ +) (m m)	0.348 (+ +) (l l)	0.3589 (+ +) (l l)	0.4177 (-) (s)	0.4239 (-) (s)	0.4345 (-) (s)	0.4460 (-) (s)	0.4322	0.45038	0.4530	0.4904
Data Class	**0.9026** (+ + +)**(l l l)**	**0.9238** (+ + +) **(l l l)**	**0.9106** (+ + +) **(l l l)**	**0.9235** (+ + +) **(l l l)**	N/A	N/A	N/A	N/A	0.267 (+ +) (m m)	0.2772 (+ +) (m m)	0.3012 (+ +) (m m)	0.3166 (+ +) (m m)	0.3249 (-) (s)	0.336 (-) (s)	0.3553 (-) (s)	0.3589 (-) (s)	0.4322	0.38158	0.3851	0.3941
Feature Envy	**0.8915** (+ + +) **(l l l)**	**0.916** (+ + + +) **(l l l)**	**0.9010** (+ + +) **(l l l)**	**0.9123** (+ + +) **(l l l)**	N/A	N/A	N/A	N/A	0.2516 (+ +) (m m)	0.2691 (+ +) (m m)	0.2880 (+ +) (m l)	0.2972 (+ +) (m l)	0.3118 (-) (s)	0.3321 (-) (s)	0.3412 (-) (s)	0.3519 (-) (s)	0.3702	0.39468	0.4012	0.4159
Long Method	**0.8832** (+ + +) **(l l l)**	**0.9041** (+ + +) **(l l l)**	**0.8906** (+ + +) **(l l l)**	**0.9056** (+ + +) **(l l l)**	N/A	N/A	N/A	N/A	0.2392 (+ +) (l l)	0.253 (+ +) (l l)	0.2517 (+ +) (l)	0.2790 (+ +) (l)	0.3053 (-) (s)	0.3311 (-) (s)	0.3308 (-) (s)	0.3212 (-) (s)	0.3401	0.37788	0.3744	0.3801
Duplicate Code	**0.8709** (+ + +) **(l l l)**	**0.8861** (+ + +) **(l l l)**	**0.8736** (+ + +) **(l l l)**	**0.8996** (+ + +) **(l l l)**	N/A	N/A	N/A	N/A	0.1964 (+ +) (m l)	0.2367 (+ +) (m l)	0.2465 (+ +) (m l)	0.2618 (+ +) (m l)	0.2759 (-) (s)	0.2891 (-) (s)	0.3112 (-) (s)	0.3386 (-) (s)	0.3270	0.32968	0.3454	0.3673
Long Parameter List	**0.8783** (+ + +) **(l l l)**	**0.9034** (+ + +) **(l l l)**	**0.9023** (+ + +) **(l l l)**	**0.9209** (+ + +) **(l l l)**	N/A	N/A	N/A	N/A	0.142 (+ +) (l l)	0.1902 (+ +) (l l)	0.1713 (+ +) (l l)	0.1814 (+ +) (l l)	0.2316 (+) (m)	0.253 (+) (m)	0.2537 (+) (m)	0.2764 (+) (m)	0.2992	0.30078	0.3170	0.3262
Spaghetti Code	**0.8649** (+ + + +) **(l l l)**	**0.8876** (+ + + +) **(l l l)**	**0.8956** (+ + + +) **(l l l)**	**0.9031** (+ + + +) **(l l l)**	0.0813 (- + +) (s l l)	0.117 (- + +) (s l l)	0.1252 (+ + +) (l l l)	0.1390 (+ + +) (l l l)	0.1130 (+ +) (m l)	0.1299 (+ +) (m l)	0.1192 (+ +) (l l)	0.1308 (+ +) (l l)	0.1903 (+) (m)	0.204 (+) (m)	0.2150 (-) (s)	0.2261 (-) (s)	0.2536	0.26408	0.2857	0.2914
Functional Decomposition	**0.8604** (+ + + +) **(l l l)**	**0.8872** (+ + + +) **(l l l)**	**0.8812** (+ + + +) **(l l l)**	**0.8932** (+ + + +) **(l l l)**	0.0635 (+ + +) (m l l)	0.0883 (+ + +) (m l l)	0.1139 (+ + +) (l l l)	0.124 (+ + +) (l l l)	0.1022 (+ +) (m l)	0.1251 (+ +) (m l)	0.1610 (+ +) (m l)	0.1842 (+ +) (l l)	0.1549 (+) (m)	0.171 (+) (m)	0.1689 (-) (s)	0.1701 (-) (s)	0.2074	0.20898	0.2309	0.2426

- The sign "+" at the i^{th} position means that the algorithm metric median value (*PF-measure_dist* (abbreviated by PF_d), *IAC*, *Precision*, or *Recall* median value) is statically different from the i^{th} algorithm value. The sign "-" means the opposite.
- The effect sizes values (small (s), medium (m), and large (l)) using the A-statistics are given.
-The N/A signifies that the given approach is Not Applicable (N/A) on the corresponding smell.
- Best obtained metrics values are highlighted in Bold. Second-best obtained metrics values are underlined.

[17]In the certain environment, the *PF-measure_dist* (abbreviated by PF_d) and *IAC* measures become *F*-measure (abbreviated by Fm) and *PCC* measures, respectively.

Table 9 *PF-measure_dist*, *IAC*, *Precision*, and *Recall* median scores of ADIPE and PDT baseline approach over 31 independent runs regarding the detection task the uncertainty levels UL=50% and UL=0%.[19]

Systems	UL	ADIPE				Baseline PDT			
		PF_d (= Fm)	IAC (= PCC)	Precision	Recall	PF_d (= Fm)	IAC (= PCC)	Precision	Recall
GanttProject	UL=50%	**0.9084** (+) (l)	**0.9288** (+) (l)	**0.9016** (+) (l)	**0.9173** (+) (l)	0.5039	0.5133	0.6087	0.6308
	UL=0%	**0.9177** (+) (l)	**0.939** (+) (l)	**0.9241** (+) (l)	**0.9173** (+) (l)	0.5158	0.5297	0.6011	0.6287
ArgoUML	UL=50%	**0.9187** (+) (l)	**0.9452** (+) (l)	**0.9004** (+) (l)	**0.9120** (+) (l)	0.5017	0.5043	0.5971	0.6118
	UL=0%	**0.9327** (+) (l)	**0.9498** (+) (l)	**0.9306** (+) (l)	**0.9120** (+) (l)	0.5216	0.52382	0.6135	0.6221
Xercess-J	UL=50%	**0.9047** (+) (l)	**0.9295** (+) (l)	**0.8985** (+) (l)	**0.9061** (+) (l)	0.4558	0.4694	0.5893	0.6012
	UL=0%	**0.9208** (+) (l)	**0.9437** (+) (l)	**0.9176** (+) (l)	**0.9061** (+) (l)	0.4712	0.4766	0.6071	0.6128
JFreeChart	UL=50%	**0.9285** (+) (l)	**0.9557** (+) (l)	**0.9174** (+) (l)	**0.9256** (+) (l)	0.4024	0.4138	0.6225	0.6419
	UL=0%	**0.9466** (+) (l)	**0.9617** (+) (l)	**0.9523** (+) (l)	**0.9256** (+) (l)	0.4102	0.4127	0.6220	0.6403
Azureus	UL=50%	**0.9277** (+) (l)	**0.9408** (+) (l)	**0.9163** (+) (l)	**0.9283** (+) (l)	0.3587	0.3736	0.6204	0.6311
	UL=0%	**0.9389** (+) (l)	**0.9548** (+) (l)	**0.9412** (+) (l)	**0.9283** (+) (l)	0.3751	0.3825	0.6305	0.6392
Apache Ant	UL=50%	**0.9165** (+) (l)	**0.9353** (+) (l)	**0.9210** (+) (l)	**0.9331** (+) (l)	0.3259	0.3515	0.6403	0.6729
	UL=0%	**0.9447** (+) (l)	**0.9577** (+) (l)	**0.9508** (+) (l)	**0.9331** (+) (l)	0.3446	0.3628	0.6697	0.6873

- The sign "+" at the i^{th} position means that the algorithm metric median value (*PF-measure_dist* (abbreviated by PF_d), *IAC*, *Precision*, or *Recall* median value) is statically different from the i^{th} algorithm value. The sign "-" means the opposite.
- The effect sizes values (small (s), medium (m), and large (l)) using the A-statistics are given.
- Best obtained metrics values are highlighted in Bold. Second-best obtained metrics values are underlined.

[19] In the certain environment, the *PF-measure_dist* (abbreviated by PF_d) and *IAC* measures become *F*-measure (abbreviated by Fm) and *PCC* measures, respectively.

the PDT ensemble generated by ADIPE returns a possibility distribution with two values for the detection case. In case of plausible smelliness, the distribution has the form [1, X], where X<1. A straightforward question that could come to the reader's mind is: "Why not only considering the value 1 and ignoring the X?" Differently speaking, "what is the added value of X?" Differently to probability theory (always for the case of detection) where the first probability value p_1 obliges the second one p_2 to be equal to (1-p_1), this influence rule does not exist in the possibility distribution because the sum of possibilities values could be greater than one. In this way, X defines the non-possibility value. This would be very helpful for the software engineer in prioritizing smelly classes for the refactoring task. Indeed, the lower the X value is, the higher the priority to apply refactoring to the considered software class is. Assuming we have the following three smelliness possibility distributions for the three java classes C_1, C_2, and C_3, respectively: [1, 0.82], [1, 0.17], and [1, 0.34]; than the refactoring should be first applied to C_2 because its non-possibility value in terms of smelliness is the lowest: 0.17. AfterC_2, the engineer should apply refactoring to C_3 (X = 0.34). Finally, the last considered class is C_1 because it has the highest value

Table 10 *PF-measure_dist*, *IAC*, *Precision*, and *Recall* median scores of ADIPE and PDT baseline approach over 31 independent runs regarding the identification task the uncertainty levels UL=50% and UL=0%.[21]

Systems	UL	ADIPE				Baseline PDT			
		PF_d (= Fm)	IAC (= PCC)	Precision	Recall	PF_d (= Fm)	IAC (= PCC)	Precision	Recall
Blob	UL=50%	**0.9228** (+) (l)	**0.923** (+) (l)	**0.9345** (+) (l)	**0.9420** (+) (l)	0.429	0.4464	0.4032	0.4175
	UL=0%	**0.9294** (+) (l)	**0.943** (+) (l)	**0.9511** (+) (l)	**0.9511** (+) (l)	0.4117	0.4331	0.4070	0.4208
Data Class	UL=50%	**0.8941** (+) (l)	**0.9207** (+) (l)	**0.9080** (+) (l)	**0.9141** (+) (l)	0.3139	0.3314	0.2962	0.3226
	UL=0%	**0.9026** (+) (l)	**0.9238** (+) (l)	**0.9235** (+) (l)	**0.9235** (+) (l)	0.3221	0.3362	0.3013	0.3291
Feature Envy	UL=50%	**0.8861** (+) (l)	**0.9037** (+) (l)	**0.8913** (+) (l)	**0.9065** (+) (l)	0.3023	0.3196	0.2907	0.3076
	UL=0%	**0.8915** (+) (l)	**0.916** (+) (l)	**0.9123** (+) (l)	**0.9123** (+) (l)	0.3098	0.3247	0.2892	0.3315
Long Method	UL=50%	**0.8794** (+) (l)	**0.896** (+) (l)	**0.8831** (+) (l)	**0.9016** (+) (l)	0.3012	0.3191	0.2823	0.3263
	UL=0%	**0.8832** (+) (l)	**0.9041** (+) (l)	**0.9056** (+) (l)	**0.9056** (+) (l)	0.3031	0.3253	0.2914	0.3119
Duplicate Code	UL=50%	**0.8623** (+) (l)	**0.8849** (+) (l)	**0.8736** (+) (l)	**0.8919** (+) (l)	0.2701	0.276	0.2647	0.2807
	UL=0%	**0.8709** (+) (l)	**0.8861** (+) (l)	**0.8996** (+) (l)	**0.8996** (+) (l)	0.2724	0.2879	0.2683	0.2847
Long Parameter List	UL=50%	**0.8721** (+) (l)	**0.897** (+) (l)	**0.8994** (+) (l)	**0.9116** (+) (l)	0.2242	0.2454	0.2158	0.2206
	UL=0%	**0.8783** (+) (l)	**0.9034** (+) (l)	**0.9209** (+) (l)	**0.9209** (+) (l)	0.2287	0.2467	0.2018	0.2235
Spaghetti Code	UL=50%	**0.8593** (+) (l)	**0.8840** (+) (l)	**0.8810** (+) (l)	**0.8952** (+) (l)	0.2118	0.2314	0.1907	0.2184
	UL=0%	**0.8619** (+) (l)	**0.8846** (+) (l)	**0.9031** (+) (l)	**0.9031** (+) (l)	0.2145	0.2258	0.1993	0.2204
Functional Decomposition	UL=50%	**0.8623** (+) (l)	**0.887** (+) (l)	**0.8724** (+) (l)	**0.8809** (+) (l)	0.1498	0.1676	0.1356	0.1497
	UL=0%	**0.8649** (+) (l)	**0.8876** (+) (l)	**0.8812** (+) (l)	**0.8932** (+) (l)	0.1532	0.1753	0.1487	0.1511

- The sign "+" at the i^{th} position means that the algorithm metric median value (*PF-measure_dist* (abbreviated by PF_d), *IAC*, *Precision*, or *Recall* median value) is statically different from the i^{th} algorithm value. The sign "-" means the opposite.
- The effect sizes values (small (s), medium (m), and large (l)) using the A-statistics are given.
- Best obtained metrics values are highlighted in Bold. Second-best obtained metrics values are underlined.

[21] In the certain environment, the *PF-measure_dist* (abbreviated by PF_d) and *IAC* measures become *F*-measure (abbreviated by Fm) and *PCC* measures, respectively.

of non-possibility in terms of smelliness. This allows ranking the detected classes in terms of smells prioritization for refactoring based on the increasing order of X values, and thus the ranking would be: C_2, C_3, and then C_1. Such ranking information defined by the X values expresses the degree of uncertainty of the human engineers and is of an important practical interest, since it allows ranking the detected smelly classes from the highest priority to the lowest one for refactoring. Moreover, in practical industrial context, the human engineer could face a high number of smelly classes and thus could be unable to process all of them. The possibility distributions offer a solution to the engineer by allowing the ranking of detected smelly classes in terms of priority for refactoring. Thus, the engineer could focus on the highest priority classes (e.g., those having X>0.65 in their possibility distributions) and keep the others for possible future investigation. This practice is very useful especially with the increase of the

software size in terms of the number of classes and also with the pressure imposed by the defined deadlines.

The identification task could be seen as a subcase of the detection one as it uses a PDT ensemble for each smell type (e.g., PDT ensemble for Blob, another one for feature envy, etc.). Therefore for each software class, after the application of the PDT detectors, we would obtain N binary possibility distributions vectors in the form $[pos_1, pos_2]$, where N is the number of smell types, pos_1 is the possibility degree of the presence of a particular smell type, and pos_2 is its non-possibility degree. In fact, the interpretation of the possibility distribution vectors by the human engineer could be a quite fastidious task, especially with the increase of the number of smell types as a distribution is generated for each smell. To solve this issue, we apply the AFO voting fusion (Dubois and Prade 1994a), as an aggregation operator, to output a single possibility distribution vector that indicates the possibility of occurrence of each considered smell type. The possibility values are used to define the priority of each smell type for refactoring. For example, assuming we have the following possibility distribution: $pos(Blob) = 1, pos(SpaghettiCode) = 0.1, pos(LongMethod) = 0.8$; for a particular class. This defines the following priority order: Blob, Long Method, and then Spaghetti Code. Likewise the detection phase, such prioritization could be very helpful for the software engineers in practical context, especially when the software size and/or the deadlines' pressure increase(s).

In summary, the possibility distributions allow prioritization for refactoring: (1) among software classes for the case of detection and (2) among smell types for the case of identification (as processing is made class per class in such task). The $PF - measure\_dist$ and IAC empirical results confirm that ADIPE's generated PDT ensembles (ADIPE detectors) outperform the considered peer methods in terms of not only detection/identification but also smells prioritization for refactoring.

5 Threats to Validity

This section explores the different factors that may skew our empirical study. These factors can be classified into three categories: (1) the internal validity, (2) the external validity and (2) the construct one. The internal validity threats concern the correctness of the results of our proposal's experiments, whereas external validity threats are associated with the generalizability of the obtained results except the sample instances employed during the experiment. Finally, the construct validity threats are related to the theory-observation relation.

5.1 Internal validity threats

During this work, we consider the internal threats to validity when using the stochastic algorithm, since 31 independent simulation runs are conducted. The peer algorithms are statically evaluated using the Wilcoxon-rank sum test with a 95% confidence level (i.e., alpha=5 %). Another important internal threat that must be tackled in our future work is the parameter configuration of the various considered optimization algorithms

through our experimentations. In this work, the parameter configuration is performed employing the trial-and-error technique (Eiben and Smit 2011), which is the widely used technique by the SBSE community. To mitigate this threat, it might be a very interesting perspective if we build a configuration strategy to update our approach parameters including the threshold parameter that is employed to decide if the current node is a leaf or internal node.

5.2 External validity threats

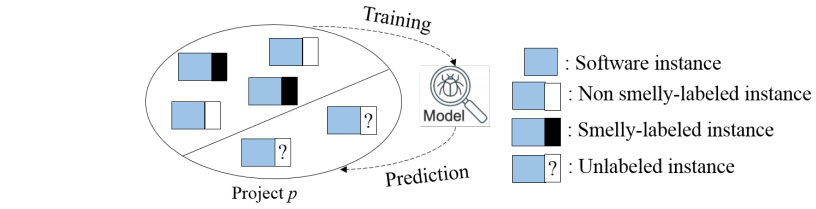
The external threat to validity mainly addresses the used types of smells in this study as well as the studied software projects. We have considered eight different types of code smells (cf. Appendix A) that are a broadly representative collection of standard smell types and the most frequent ones. We have chosen six different software projects (cf. Table 2) from various application domains, having different sizes as well as diverse functionalities, and also built by different companies. The choice of such diverse projects aims to reduce the bias that could appear due to the special characters on the selected projects. Moreover, we carried out a K-folds cross-validation strategy to reduce the bias evoked by the specific projects. Besides, it is very interesting if we could test our proposed ADIPE tool for the identification of anti-patterns that reside on web services and Android applications. For the case of web services, anti-patterns may impede their progress, thereby deteriorating their quality and automatically decreasing the rate of use. On the Android applications side, the existence of anti-patterns will negatively affect these applications' executions by raising their processing times and also by increasing their consumption in terms of energy. Figure 18 shows the detection code smells models on a within-project and a cross-project. One can see from Figure 18 (a) that in the case of a within-project, the unseen software classes are labeled based on the trained instances of the source projects considered in the training phase. However, in the case of a cross-project (cf. Figure 18 (b)), the unseen classes of the target project are labeled by the help of the labeled instances of the source project and some of its labeled instances (cf. Appendix H). Based on Figure 18, it will be interesting to add a transfer learner to our ADIPE approach that is able to transfer knowledge of the considered source projects to target projects.

5.3 Construct validity threats

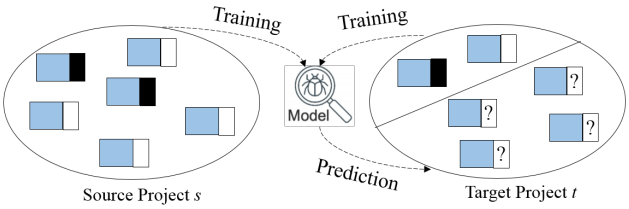
In our experimental study, we have built a BE using some existing advisors (e.g., DECOR (Moha et al. 2010), JDeodorant (Tsantalis and Chatzigeorgiou 2009), inFusion[22], iPlasma[23], and PMD (Gopalan 2012)) for the detection of anti-patterns (cf. Figure 7). Then, the produced BE is submerged by uncertainty, in particular, its class labels. The transformation from crisp (or certain) class labels to uncertain ones is made based on: (1) five different probabilistic classifiers ((Naïve Bayes classifier (Friedman et al. 1997), Probabilistic K-NN (Holmes and Adams 2002), Bayesian

[22]http://www.intooitus.com/products/infusion

[23]http://loose.upt.ro/iplasma/



(a) Within-project Code smell detection



(b) Cross-project Code smell detection

Fig. 18 Within-project and Cross-project Code smell detection (Inspired by (Zhu et al. 2020))

.

Networks (Pearl 1982, 1985), Naïve Bayes Nearest Neighbor (Behmo et al. 2010), and Probabilistic Decision Tree (Quinlan 1987)) for the creation of the likelihood values and then (2) a mathematical formula to transform the obtained probability distributions into a possibilistic ones (for more details, please refer to Section 3.1). Thereby, a constructed threat to validity can be related to the employment of the non-deterministic five probabilistic classifiers for the simulation of the uncertain as well as subjective experts' opinions in the process of the generation of likelihood values. To handle such a problem, the generated values will be manually checked by human experts. To the best of our knowledge, our work is the first one in the SBSE that detects code smells with uncertain class labels. Based on this fact, another important construct threat to validity arises because there is no SBSE work that dealt with the detection of code smells under uncertain environment. Unfortunately, most of the current approaches ignore the uncertainty that resides in the BE. So, we compared our approach with a possibilistic one (i.e., NS-PDT). This latter is not publicly available and hence we re-implemented it. However, the NS-PDT re-implementation could be wrong, and this could skew the obtained results. In the aim to reduce the threat, we relied on (experienced) code reviewers to monitor implementation. We also compared with the reported results of the re-implemented NS-PDT in the literature, and therefore, the comparison shows that the results are nearly identical. For the evaluation part, it is impossible to evaluate the performance of our proposed approach against the remaining ones using the existing measures in the literature since these latter are maybe able to work under an imbalanced environment, but not for the uncertain environment. We recall that in this work, we are dealing with an uncertain environment as well as an imbalanced one. To handle these issues, we have proposed a novel metric called *PF-measure_dist*, which can work under certain and uncertain environments. To the best of our knowledge the *PF-measure_dist* is the first uncertain metric that has been

proposed in the SBSE field. Thus, an important construct threat to validity may appear due the lack of existence of uncertain measures. In fact, the usual used metrics (i.e., *F-measure*, *Accuracy*, *AUC*, etc.) are not suitable to deal with the uncertainty over the class labels. For this reason, we have used the *PF-measure_dist* and the *IAC* measures to consider the information about the uncertainty existing at the class labels. Although, we have employed the *AUC* metric to compare the five considered algorithms in the certain environment for the detection and identification cases (cf. Appendix E), it will be interesting to investigate the performance of ADIPE using additional metrics such as the Geometrical mean (G-mean) and the modified Area Under Precision Recall-One Versus All (mAURPC-OVA).

6 Related work

The detection of code smells remains a highly active and timely subject for research within the SE domain, including the SBSE one (Azeem et al. 2019). Many types of research have been suggested by different authors to automate the code smells detection methods to aid the experts (including developers) for the detection task. The term uncertainty has appeared in some SE problems, (Whittle et al. 2009) tackled the uncertainty caused by changing environmental conditions for the case of self-adaptive systems. Also, the uncertainty was discussed by (Bowers et al. 2020) where they try to optimize the non-functional requirements in the self-adaptive system that generates adaptation strategies for carrying out reconfigurations at run time to tackle unexpected problems that occur as a result of uncertainty such as, unpredicted problems within the system itself. Nevertheless, most of the existing works did not deal with the uncertain data in the SE including the SBSE methods that can be classified into four major categories: (1) Rule-based methods, (2) Machine learning-based methods (along with deep learning), (3) Search-based methods, and (4) Others. Table 11 summarizes the main features and the hyper-parameters of the most prominent existing works, with the aim to show the characteristics that make ADIPE able to deal with uncertain and imbalanced data with respect to existing works. Based on Table 11, the following observations could be concluded:

- ADIPE is the only search-based algorithm that is able to take into consideration the doubtfulness and the subjectivity of the software engineers when tackling the detection of code smells problem. This is done in the construction of the BE phase through the use of a set of Opinion-based classifiers that generate probabilistic values (i.e., likelihood values) to take into consideration the uncertainty of the software engineer. These latter are converted into possibilistic values in order to mimic real world scenarios.
- Most of the existing works have not consider the data imbalance issue. Only two works proposed to tackle this issue by focusing in the data level by means of data processing (Fontana et al. 2016b; Di Nucci et al. 2018). In fact, these two works employed stratified random sampling to rebalance data. ADIPE is the only work that propose to handle the imbalance data problem by focusing on the algorithm level through the use of a fitness function that is based on the *PF-measure_dist*. It

is important to note that the *PF-measure_dist* is inspired by the *F-measure* metric, which is insensitive to the data imbalance issue.
- Existing DTs based methods use the gain ratio as feature selection criterion at each node. This practice is greedy and could lead to the local optimal split. Other researchers proposed to use the random forest where a set of DTs are constructed using a random selection of the feature before aggregating all the generated trees. This random generation process may also lead to locally-optimal splits. In ADIPE, at each node of PDT, only one feature is chosen using the crossover and mutation operators. On the one hand, the crossover exploits the fittest parts of the parents with the aim to generate prominent offspring individuals. On the other hand, the mutation operator is used to diversify the offspring individuals (i.e., PDTs) in order to not get stack in local optima. Therefore, our ADIPE performs an effective feature selection at PDT nodes based on: (1) the crossover and mutation operators to perform the feature selection operation and (2) the *PF-measure_dist* towards the global-optimal PDT.
- The column TDM presents the efficient threshold generation carried out by ADIPE using the Kretowski-&-Grzes method. In fact, existing machine learning-based approaches have used , the entropy-based discretization method for the threshold generation. This method is greedy and usually leads to locally-optimal thresholds. For this reason, researchers have opted for the use of the mutation operator. This latter have shown its limitation since it can leads to the generation of meaningless and/or ineffective splitting thresholds. In this work, we have chosen the Kretowski-&-Grzes method for the following two main reasons. On the one hand, the application of this method at each node of the PDT produces a number of effective thresholds for the related feature (cf. Figure 10). On the other hand, ADIPE chooses randomly a threshold from the set of effective splitting thresholds generated based on the Kretowski-&-Grzes method to avoid being trapped in local optimal thresholds (which is not the case of C4.5).
- Traditional existing detection methods employ two types of pruning. The first one consists on pruning the nodes during the induction, while the second one prunes them after the whole tree generation. The SBSE approaches (including ADIPE) perform the pruning process by applying the crossover operator. During the evolution process, the crossover operator could stop the development of branches via the sub-tree exchange.
- Column CEFF on the table shows that many fitness functions have been employed to optimize the detection rules. Nevertheless, no one of them considers the uncertainty and the data imbalance issues. ADIPE uses the *PF-measure_dist* as a fitness function. This latter has proven its ability to deal with the uncertainty and its insensitivity to the problem of imbalanced data.

6.1 Rule/heuristic-based approaches

The initial efforts for the identification of software classes containing anti-patterns focused on defining rule-based methods (also called heuristic-based approaches) (Sharma and Spinellis 2018) that relies on structural metrics for capturing deviations

Table 11 Comparison between the existing approaches of code smells detection on the basis of their features and hyper-parameters (IEC: Imbalance Environment Consideration, UEC: Uncertain Environment Consideration, CM: Classifier Model, ASU: Adaptation Strategy to Uncertainty, NFSC: Node Feature Selection Criterion, TDM: Threshold Definition Method, PS: Pruning Strategy, CEFF: Classifier Evaluation Fitness Function, , N/A: Not Applicable , N/R: Not Reported).

Category	Smell detection method	IC		UC		CM	ASU	NFSC	TDM	PS	CEFF
		No	Yes	No	Yes						
Rules/heuristic-based approaches	(Marinescu 2002)	x		x		Heuristic rules are defined manually	Nothing	N/A	N/A	N/A	N/A
	(Moha et al. 2010)	x		x		Heuristic rules are defined manually	Nothing	N/A	N/A	N/A	N/A
	(Tsantalis and Chatzigeorgiou 2011)	x		x		Heuristic rules are defined manually	Nothing	N/A	N/A	N/A	N/A
Machine learning-based approaches	(Kreimer 2005)	x		x		Decision tree (C4.5)	Nothing	Gain Ratio	Entropy relying on discretization	Pre-pruning based on Gain Ratio	N/A
	(Amorim et al. 2015)	x		x		Decision tree (C5.0)	Nothing	Gain Ratio	Entropy relying on discretization	Pre-pruning based on Gain Ratio	N/A
	(Khomh et al. 2009)	x		x		Belief Bayesian Network	Nothing	N/A	N/A	N/A	N/A
	(Khomh et al. 2011)	x		x		Belief Bayesian Network	Nothing	N/A	N/A	N/A	N/A
	(Vaucher et al. 2009)	x		x		Belief Bayesian Network	Nothing	N/A	N/A	N/A	N/A
	(Maiga et al. 2012a,b)	x		x		Support Vector Machine	Nothing	N/A	N/A	N/A	N/A
	(Hassaine et al. 2010)	x		x		Artificial Immune System based classifier	Nothing	N/A	N/A	N/A	N/A
	(Oliveto et al. 2010)	x		x		B-Splines based classifier	Nothing	N/A	N/A	N/A	N/A
	(Fontana et al. 2016b)		x	x		Pruned Decision tree (C4.5 without Boosting and C4.5 with Boosting)	Nothing	Gain Ratio	Entropy based on discretization	Pre-pruned based on Gain Ratio	N/A
						Unpruned Decision tree (C4.5 without Boosting and C4.5 with Boosting)				Nothing	
						Unpruned Decision tree (C4.5 without Boosting and C4.5 with Boosting)				Post-pruning based on Error Reduction	
						JRip classifier (RIPPER with Boosting and RIPPER without Boosting		N/A	N/A	Post-pruning based on Error Reduction	
						Random Forest (Random construction with Boosting and Random construction without Boosting)		Arbitrary Feature selection	N/A	Nothing	
						Naïve Bayes with (and without) Boosting		N/A	N/A	N/A	
						SVM (SMO (RBF and Poly) without Boosting and S MO (RBF and Poly) with Boosting		N/A	N/A	N/A	
						Linear, Polynomial, Radial, and Sigmoid SVM variances with Boosting and without		N/A	N/A	N/A	
	(Di Nucci et al. 2018)		x	x		Similar classifiers as adopted by (Fontana et al. 2016b)	Nothing				

Category	Smell detection method	IC		UC		CM	ASU	NFSC	TDM	PS	CEFF
		No	Yes	No	Yes						
Search-based approaches	(Kessentini et al. 2011)	x		x		Tree of Ad-hoc Rules	Nothing	Crossover-Mutation	Mutation	Crossover	NB_{TP} [a]
	(Ouni et al. 2013)	x		x		Tree of Ad-hoc Rules	Nothing	Crossover-Mutation	Mutation	Crossover	F_{norm} [b]
	(Boussaa et al. 2013)	x		x		Tree of Ad-hoc Rules	Nothing	Crossover-Mutation	Mutation	Crossover	$F_{coverage}$ and $Cost$ [c]
	(Kessentini et al. 2014)	x		x		Tree of Ad-hoc Rules	Nothing	Crossover-Mutation	Mutation	Crossover	F_{GP} and F_{GA} [d]
	(Sahin et al. 2014)	x		x		Tree of Ad-hoc Rules	Nothing	Crossover-Mutation	Mutation	Crossover	F_{upper} and F_{lower} [e].
	(Mansoor et al. 2017)	x		x		Tree of Ad-hoc Rules	Nothing	Crossover-Mutation	Mutation	Crossover	F_1 and F_2 [f].
	(Boutaib et al. 2021)		x		x	Possibilistic K-NN	Possibilistic K-NN evolution using PG-mean	Crossover-Mutation	Mutation	Crossover	*PG-mean* [g]
	Our proposed ADIPE		x		x	Possibilistic Decision Tree	Possibilistic Decision Tree evolution using PF-measure_dist	Crossover-Mutation	Krętowski & Grześ Method	Crossover	*PF-measure_dist*
Others	(Rapu et al. 2004)	x		x		Manual defined heuristic rule	Nothing	N/A	Manual conveived heuristic choice	N/A	N/A
	(Palomba et al. 2013, 2015)	x		x		Manual defined heuristic rule	Nothing	N/A	Manual conveived heuristic choice	N/A	N/A
	(Fu and Shen 2015)	x		x		Manual defined heuristic rule	Nothing	N/A	Manual conveived heuristic choice	N/A	N/A
	(Emden and Moonen 2002)	x		x		Manual defined heuristic rule	Nothing	N/A	Manual conveived heuristic choice	N/A	N/A
	(Langelier et al. 2005)	x		x		Manual defined heuristic rule	Nothing	N/A	Manual conveived heuristic choice	N/A	N/A
	(Dhambri et al. 2008)	x		x		Manual defined heuristic rule	Nothing	N/A	Manual conveived heuristic choice	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)

[g] $PG-mean = \sqrt{Sensitivity\_dist \times Specificity\_dist}$. The $PG-mean$ is the geometrical mean of the Specificity and Sensitivity over the minority class labels in the uncertain environment (Boutaib et al. 2021).

from good practices of OO design. Erni and Lewerentz (1996) proposed to use multi-metrics to evaluate the software performance with a view of improving them. Marinescu (2004) proposed a strategy for detecting defects into different levels of OO design fragments (including method, class, and subsystem) using a metric-based approach to analyze the code source. Lanza and Marinescu (2007b) suggested to combine a number of quality metrics with thresholds where a set of a rules (i.e., a combination of AND/OR operators) are established for 11 anti-patterns. Moha et al. (2010) introduced the DECOR approach that described the symptoms of the defect using an abstract language to represent the rules. Other existing approaches have adopted clustering methods for detecting code smells such as the JDeodorant tool Tsantalis and Chatzigeorgiou (2009). This latter could detect smells and recommend some move method as refactoring operations.

6.2 Machine learning-based approaches

To tackle the manual design difficulties, a new trend has been recently emerging which consists in the employment of machine learning techniques for the problem of detection of code smells (Fontana et al. 2016b). The idea was to build a given classifier based on training data so that the classifier can predict the smelly software classes on software projects. Kreimer (2005) suggested the application of Decision Trees (DTs) methods on two small software systems (IYC and WEKA) to find the two design flaws occurrences: God Class and Long Method. Amorim et al. (2015) validated the previous results by evaluating the DTs performances on four medium software systems sizes to detect 12 different anti-patterns. Khomh et al. (2009) proposed to use Bayesian Networks for the detection of Blob anti-pattern occurring on two software projects. Khomh et al. (2011) extended their previous work by proposing a novel approach called BDTEX (Bayesian Detection Expert) that relies the Goal Question Metric to construct Belief Bayesian Networks. Vaucher et al. (2009) used the BBNs to study the Blob evolution life cycle and thereby distinguishing between the real God Classes from accidental ones. Maiga et al. (2012a,b) introduced an SVMDetect approach that uses the Support Vector Machine (SVM) method to detect anti-patterns. Hassaine et al. (2010) used the immune-inspired approach to recognize the Blob smell types. This approach is designed for systematically detecting code smells within classes that violate the characteristics belonging to certain conceived rules. Oliveto et al. (2010) suggested to use the ABS (Anti-pattern identification using B-Splins) approach in order to identify some smelly instances using the numerical analysis method.

Some authors have performed various studies to compare the performance of different machine learning techniques for the code smell detection problem. For instance, Fontana et al. (2016b) have tested and compared 16 supervised machine learning methods with their boosting variant for detecting a set of code smells on 74 open-source software projects with different scales. Furthermore, during the training as well as the evaluation stages, the authors employed the under-sampling method to avoid the poor performances generated by machine learning methods for the case of imbalanced datasets. Fontana and Zanoni (2017) made a classification of code smells based on their severity using multinomial classification and regression techniques.

This proposed approach can help developers to prioritize the class or methods or to rank them. Di Di Nucci et al. (2018) reported the limitations of Fontana et al. (2016b) approach. Recently, researchers proposed to employ the Deep Neural Network (DNN) to detect code smells using different type of information: (1) Structural and historical information (Barbez et al. 2019), (2) Structural as and textual information (Liu et al. 2019), and (3) Structural and semantic information (Hadj-Kacem and Bouassida 2019).

6.3 Search-based approaches

The search-based methods are used to address various optimization problems in the context of software engineering based on meta-heuristic algorithms. Some researchers considered the detection task as the most critical step since it records the exiting smells as well as their paths in a given software system to the next step, which corresponds to the refactoring (correction) task. Kessentini et al. (2011) introduced an automated approach that relies on detection rules to detect code smells in a given software project. The detection rules are described as combinations of quality metrics along with thresholds that have been drawn from the comparison of different heuristic search algorithms devoted to the extraction of rules. Ouni et al. (2013) implemented a search-based method to detect the existing code smells in open-source software systems. This was the first method to derive the rules for detection based on genetic programming from the smelly examples. Boussaa et al. (2013) suggested to adopt an approach that relies on a competitive co-evolutionary search to address the code smell detection problem. This approach consists of two concurrent populations that grow at the same time; the first population generates some detection rules to maximize the detection ratio for smelly examples. However, the second population maximizes the generation of artificially code smells that cannot be detected by the first population. Kessentini et al. (2014) suggested to integrate the parallel aspect by proposing a method called PEA (Parallel Evolutionary Algorithm) for the detection of code smells. In fact, the authors joined the GA as well as the GP (Genetic Programming) in a parallel manner over the optimization process to build a range of detection rules based on examples of different smell types as well as the detection from the non-smelly code source examples. Sahin et al. (2014) introduced the BLOP (Bi-Level Optimization Problem) method based on a bi-level optimization problem to produce the detection rules of the code smells across two levels. The first level is known as the upper level, and it has to generate a collection of detection rules to optimize the coverage of real code smell examples as well as the artificially generated ones across the second level. In contrast, the other level (second or the follower one) is responsible for divulging the maximum smells that are not detected from the first level. Mansoor et al. (2013) proposed MOGP (Multi-Objective Genetic Programming) method that relies on the multi-objective aspect for the detection rules generation task. MOGP is used to figure out the best quality metric combination by maximizing the detected smelly examples number as well as minimizing the number of detected well-designed examples. Boutaib et al. (2021) have proposed ADIPOK (Anti-pattern Detection and Identification using Possibilistic K-NN Evolution) that evolves a set

of PK-NN detectors to detect and/or identify code smells under certain and uncertain environments.

6.4 Others

In the last few years, researchers paid more attention to historical information for the code source evolution to detect code smells. Rapu et al. (2004) proposed to extract the historical information from the smelly structure. The authors considered some historical measures that depict the evolution of smells. Palomba et al. (2013, 2015) developed a novel approach namely HIST (Historical Information for Smell deTection). This approach identifies code smells based on the extracted historical information from different software versions. For the evaluation of the HIST approach, the authors used eight software systems and a set of smell types. Fu and Shen (2015) suggested an approach that relies on associated rule mining to draw information history software systems with an enormous growth history. Indeed, the information history could be related to either classes or methods or packages as well as to an additional operation or modification one. Emden and Moonen (2002) introduced jCOSMO to visualize the code smells for complex software analysis. This latter involves parsing source code as well as showing the smelly code fragments and their relation via the graphical view. Langelier et al. (2005) introduced the VERSO framework that used colors in the visualization in order not only to represent the properties but also to support the quality analysis in a given software. Dhambri et al. (2008) developed an approach that automatically detects some symptoms of code smells and keeps the rest of the analysis for a human analyst.

7 Conclusion and future work

In this paper, we proposed ADIPE as a new method and tool for detecting and identifying code smells for uncertain and certain environments in which uncertainty mainly appears at the level of class labels. Over this original work, we have triggered a novel trend that the SBSE community researchers often neglect; which corresponds to the fact that the code smells detection and/or identification is an uncertain classification problem. In fact, the uncertainty could be justified by the subjectivity and the doubtfulness that software engineers usually face when determining not only the smelliness classes but also the smell types that they include. Additionally, the human software engineers may have diverse opinions since they have distinct knowledge as well as expertise. Ignoring any amount of uncertainty could lead to a significant loss of information and consequently diminishes the performance of whenever used detectors. To deal with the uncertainty issue, ADIPE evolves a set of PDTs classifiers using a PBE. Our proposal has proven its merits by dint of three important characteristics. First, it employs an adequate detector (PDT classifier) that is suitable to deal with uncertain class labels. Moreover, this kind of classifier is evolved using the GA aiming to escape local optima and to approach the global optima. We recall that over this work we adopted the possibility theory as a tool to model the uncertainty through possibility

distributions. Second, the designed fitness function (*PF*-mesure_dist) is capable to manage the uncertain data as it takes into consideration the possibility distributions and the imbalance issue due to its proven insensitivity. Finally, the AFO (Adaptive Fusion Operator) is used since it is a suitable fusion operator to merge the different possibility distributions coming from different detectors. The specificity of the AFO relies on its ability to merge conflictual as well as non-conflictual information (i.e., possibility distribution) at the same time. Our proposed decision tool has an industrial impact for the following two main reasons: (1) it is able to maintain several industrial source projects, and (2) it helps the software engineer to made effective choices regarding the refactoring operations sequences that should be applied.

As part of our perspectives, we plan to expand our BE by generating artificial code smells that might not be covered in our study. Moreover, we aim to evaluate our ADIPE against the remaining PDT types: Sim-PDT (Jenhani et al. 2008a) and Clust-PDT (Jenhani et al. 2009). Also, we may face a problem where an important amount of data are not labeled. To this end, we propose to adopt one of the semi-supervised techniques that can effectively deal with the lack of data labeled problem. It would be interesting to extend this problem into an uncertain environment. We also intend to merge different information types (structural and historical (Palomba et al. 2015)) and consider learning classifiers to detect and/or identify smells under uncertain environment. In our experimental study, we have evaluated the performance of ADIPE on unseen software classes from the six software projects employed in the training phase. However, we have not predict the labels coming from target unseen projects (Zimmermann et al. 2009; Li et al. 2020). Thus, it will be interesting to equip our ADIPE with a transfer learner that is able to transfer knowledge of the considered source projects to target projects coming from different domains (Qing et al. 2015; Du et al. 2019; Zhu et al. 2020). As Future Avenue, it would be interesting to use a distributed GPU architecture in order to decrease the time complexity of the detectors generation process (cf. Appendix G). Moreover, we could add a posterior Interactive Development Environment that allows introducing the developer preferences by modifying some generated PDTs. Finally, we plan to use ADIPE for the identification of smell types residing on mobile and web-based applications (Bessghaier et al. 2020; Saidani et al. 2020).

Conflict of interest

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

Acknowledgements 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(2):205–228

Alcalá-Fdez J, Fernández A, Luengo J, Derrac J, García S, Sánchez L, Herrera F (2011) Keel data-mining software tool: data set repository, integration of algorithms and experimental analysis framework. Journal of Multiple-Valued Logic & Soft Computing 17

Amor NB, Benferhat S, Elouedi Z (2004) Qualitative classification and evaluation in possibilistic decision trees. In: Proceedings of the IEEE International Conference on Fuzzy Systems, IEEE, vol 2, pp 653–657

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,, IEEE, pp 261–269

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(3):219–250

Azeem MI, 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

Barbez A, Khomh F, Guéhéneuc YG (2019) Deep learning anti-patterns from code metrics history. In: Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME 2019), IEEE, pp 114–124

Barros RC, Basgalupp MP, De Carvalho AC, Freitas AA (2012) A survey of evolutionary algorithms for decision-tree induction. IEEE Transactions on Systems, Man, and Cybernetics 42(3):291–312

Behmo R, Marcombes P, Dalalyan A, Prinet V (2010) Towards optimal naive bayes nearest neighbor. In: European conference on computer vision, Springer, pp 171–184

Bessghaier N, Ouni A, Mkaouer MW (2020) On the diffusion and impact of code smells in web applications. In: International Conference on Services Computing, Springer, pp 67–84

Bouchon-Meunier B, Dubois D, Godo L, Prade H (1999) Fuzzy sets in approximate reasoning and information systems, vol 5. Kluwer Academic Publishers

Bounhas M, Hamed MG, Prade H, Serrurier M, Mellouli K (2014) Naive possibilistic classifiers for imprecise or uncertain numerical data. Fuzzy Sets and Systems 239:137–156

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

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

Boutaib MS, Elouedi Z (2018) Incremental possibilistic decision trees in non-specificity approach. In: Proceedings of the 13th International FLINS Conference (FLINS 2018), World Scientific, vol 11, p 339

Boutaib S, Bechikh S, Palomba F, Elarbi M, Makhlouf M, Said LB (2020) Code smell detection and identification in imbalanced environments. Expert Systems with Applications 166:114076

Boutaib S, Elarbi M, Bechikh S, Hung CC, Said LB (2021) Software anti-patterns detection under uncertainty using a possibilistic evolutionary approach. In: EuroGP, pp 181–197

Bowers KM, Fredericks EM, Hariri RH, Cheng BH (2020) Providentia: Using search-based heuristics to optimize satisficement and competing concerns between functional and non-functional objectives in self-adaptive systems. Journal of Systems and Software 162:1–51

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

Bryton S, e Abreu FB, Monteiro M (2010) Reducing subjectivity in code smells detection: Experimenting with the long method. In: 2010 Seventh International Conference on the Quality of Information and Communications Technology, IEEE, pp 337–342

Catolino G, Palomba F, Fontana FA, De Lucia A, Zaidman A, Ferrucci F (2019) Improving change prediction models with code smell-related information. arXiv preprint arXiv:190510889

Chidamber SR, Kemerer CF (1994) A metrics suite for object oriented design. IEEE Transactions on software engineering 20(6):476–493

Conover WJ, Conover WJ (1980) Practical nonparametric statistics. Wiley New York

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

Di Nucci D, Palomba F, Tamburri DA, 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, IEEE, pp 612–621

Du X, Zhou Z, Yin B, Xiao G (2019) Cross-project bug type prediction based on transfer learning. Software Quality Journal pp 1–19

Dubois D, Prade H (1985) Unfair coins and necessity measures: towards a possibilistic interpretation of histograms. Fuzzy sets and systems 10(1-3):15–20
Dubois D, Prade H (1988) Possibility theory: an approach to computerized processing of uncertainty
Dubois D, Prade H (1994a) La fusion d'informations imprécises. Traitement du signal 11(6):447–458
Dubois D, Prade H (1994b) Possibility theory and data fusion in poorly informed environments. Control Engineering Practice 2(5):811–823
Dubois D, Prade H (2000) Possibility theory in information fusion. In: Proceedings of the 3rd international conference on information fusion, IEEE, vol 1, pp 6–P19
Dunford N, Schwartz J, WG B, RG B (1971) Linear Operators. Wiley-Interscience, New York
Eiben AE, Smit SK (2011) Parameter tuning for configuring and analyzing evolutionary algorithms. Swarm and Evolutionary Computation 1(1):19–31
Emden EV, Moonen L (2002) Java quality assurance by detecting code smells. In: Proceedings of the 9th Working Conference on Reverse Engineering,, IEEE, pp 97–106
Erni K, Lewerentz C (1996) Applying design-metrics to object-oriented frameworks. In: Proceedings of the 3rd international software metrics symposium, IEEE, pp 64–74
Fernandes E, Oliveira J, Vale G, Paiva T, Figueiredo E (2016) A review-based comparative study of bad smell detection tools. In: Proceedings of the 20th Conference on Evaluation and Assessment in Software Engineering, ACM, p 18
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 FA, Zanoni M (2017) Code smell severity classification using machine learning techniques. Knowledge-Based Systems 128:43–58
Fontana FA, Braione P, Zanoni M (2012) Automatic detection of bad smells in code: An experimental assessment. Journal of Object Technology 11(2):5–1
Fontana FA, 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), IEEE, vol 1, pp 609–613
Fontana FA, Mäntylä MV, Zanoni M, Marino A (2016b) Comparing and experimenting machine learning techniques for code smell detection. Empirical Software Engineering 21(3):1143–1191
Foundation AS (2004) Apache commons cli. URL `http://commons.apache.org/cli/`, [Accessed 19-April-2021]
Fowler M, Beck K (1999) Refactoring: Improving the Design of Existing Code. Addison-Wesely
Friedman N, Geiger D, Goldszmidt M (1997) Bayesian network classifiers. Machine learning 29(2-3):131–163
Fu S, Shen B (2015) Code Bad Smell Detection through Evolutionary Data Mining. In: Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement,, IEEE, pp 1–9
Gopalan R (2012) Automatic detection of code smells in java source code. PhD thesis, University of Western Australia
Hadj-Kacem M, Bouassida N (2019) Deep representation learning for code smells detection using variational auto-encoder. In: Proceedings of the International Joint Conference on Neural Networks (IJCNN), IEEE, pp 1–8
Hassaine S, Khomh F, Guéhéneuc YG, 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,, IEEE, pp 343–348
Henderson-Sellers B (1995) Object-oriented metrics: measures of complexity. Prentice-Hall, Inc.
Higashi M, Klir GJ (1983) On the notion of distance representing information closeness: Possibility and probability distributions. International Journal of General System 9(2):103–115
Holland JH (1992) Genetic algorithms. Scientific american 267(1):66–73
Holmes C, Adams N (2002) A probabilistic nearest neighbour method for statistical pattern recognition. Journal of the Royal Statistical Society: Series B (Statistical Methodology) 64(2):295–306
Hosseini S, Turhan B, Mäntylä M (2018) A benchmark study on the effectiveness of search-based data selection and feature selection for cross project defect prediction. Information and Software Technology 95:1–17
Hüllermeier E, Beringer J (2006) Learning from ambiguously labeled examples. Intelligent Data Analysis 10(5):419–439
Jenhani I (2010) From possibilistic similarity measures to possibilistic decision trees. PhD thesis, Artois

Jenhani I, Amor NB, Elouedi Z, Benferhat S, Mellouli K (2007) Information affinity: A new similarity measure for possibilistic uncertain information. In: Proceedings of the European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty, Springer, pp 840–852

Jenhani I, Amor NB, Benferhat S, Elouedi Z (2008a) Sim-pdt: A similarity based possibilistic decision tree approach. In: Proceedings of the International Symposium on Foundations of Information and Knowledge Systems, Springer, pp 348–364

Jenhani I, Amor NB, Elouedi Z (2008b) Decision trees as possibilistic classifiers. International Journal of Approximate Reasoning 48(3):784–807

Jenhani I, Benferhat S, Elouedi Z (2009) On the use of clustering in possibilistic decision tree induction. In: Proceedings of the European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty, Springer, pp 505–517

Karafotias G, Hoogendoorn M, Eiben ÁE (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,, Springer, vol 6603, pp 401–415

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(9):841–861

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

Khomh F, Vaucher S, Guéhéneuc YG, Sahraoui H (2011) BDTEX: A GQM-based Bayesian approach for the detection of antipatterns. Journal of Systems and Software 84(4):559–572

Klement EP, Mesiar R, Pap E (2000) Triangular norms

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

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

Kroupa T (2006) Application of the choquet integral to measures of information in possibility theory. International journal of intelligent systems 21(3):349–359

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,, ACM, pp 214–223

Lanza M, Marinescu R (2007a) Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems. Springer Science & Business Media

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

Li K, Xiang Z, Chen T, Tan KC (2020) Bilo-cpdp: Bi-level programming for automated model discovery in cross-project defect prediction. In: 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, pp 573–584

Liu H, Jin J, Xu Z, Bu Y, Zou Y, Zhang L (2019) Deep learning based code smell detection. IEEE Transactions on Software Engineering

Ma CY, Wang XZ (2009) Inductive data mining based on genetic programming: Automatic generation of decision trees from data for process historical data analysis. Computers & Chemical Engineering 33(10):1602–1616

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

Maiga A, Ali N, Bhattacharya N, Sabané A, Guéhéneuc YG, 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,, IEEE, pp 278–281

Mansoor U, Kessentini M, Bechikh S, Deb K (2013) Code-smells detection using good and bad software design examples. Technical report, Technical Report

Mansoor U, Kessentini M, Maxim BR, Deb K (2017) Multi-objective code-smells detection using good and bad design examples. Software Quality Journal 25(2):529–552

Mantyla MV, Vanhanen J, Lassenius C (2004) Bad smells-humans as code critics. In: 20th IEEE International Conference on Software Maintenance, 2004. Proceedings., IEEE, pp 399–408
Marinescu R (2002) Measurement and quality in object oriented design. PhD 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, IEEE, pp 350–359
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, IEEE, pp 274–275
Martin RC (2002) Agile software development: principles, patterns, and practices. Prentice Hall
McConnell S (2004) Code Complete - A Practical Handbook of Software Construction. Microsoft Press
Moha N, Gueheneuc YG, Duchien L, Meur AFL (2010) DECOR: A Method for the Specification and Detection of Code and Design Smells. IEEE Transactions on Software Engineering 36(1):20–36
Oliveto R, Khomh F, Antoniol G, Guéhéneuc YG (2010) Numerical signatures of antipatterns: An approach based on b-splines. In: Proceedings of the 14th European Conference on Software maintenance and reengineering,, IEEE, pp 248–251
Ouni A (2014) A mono-and multi-objective approach for recommending software refactoring. PhD 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(1):47–79
Padhye N, Mittal P, Deb K (2015) Feasibility preserving constraint-handling strategies for real parameter evolutionary optimization. Computational Optimization and Applications 62(3):851–890
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, Bavota G, Penta MD, Oliveto R, Lucia AD, 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, IEEE Press, pp 268–278
Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A (2014) Do they really smell bad? a study on developers' perception of bad code smells. In: 2014 IEEE International Conference on Software Maintenance and Evolution, IEEE, pp 101–110
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(5):462–489
Palomba F, Panichella A, De Lucia A, Oliveto R, Zaidman A (2016) A textual-based technique for smell detection. In: Proceedings of the 24th international conference on program comprehension (ICPC), IEEE, pp 1–10
Palomba F, Panichella A, Zaidman A, Oliveto R, De Lucia A (2017a) The scent of a smell: An extensive comparison between textual and structural smells. IEEE Transactions on Software Engineering 44(10):977–1000
Palomba F, Zanoni M, Fontana FA, De Lucia A, Oliveto R (2017b) Toward a smell-aware bug prediction model. IEEE Transactions on Software Engineering
Palomba F, Bavota G, Di Penta M, Fasano F, Oliveto R, De Lucia A (2018a) On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. Empirical Software Engineering 23(3):1188–1221
Palomba F, Tamburri DAA, Fontana FA, Oliveto R, Zaidman A, Serebrenik A (2018b) Beyond technical aspects: How do community smells influence the intensity of code smells? IEEE transactions on software engineering
Palomba F, Zaidman A, De Lucia A (2018c) Automatic test smell detection using information retrieval techniques. In: Proceedings of the International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp 311–322
Pan SJ, Tsang IW, Kwok JT, Yang Q (2010) Domain adaptation via transfer component analysis. IEEE transactions on neural networks 22(2):199–210
de Paulo Sobrinho EV, De Lucia A, de Almeida Maia M (2018) A systematic literature review on bad smells—5 w's: which, when, what, who, where. IEEE Transactions on Software Engineering
Pearl J (1982) Reverend bayes on inference engines: A distributed hierarchical approach. In: Proceedings of the Second AAAI Conference on Artificial Intelligence, AAAI Press, p 133–136
Pearl J (1985) Bayesian netwcrks: A model cf self-activated memory for evidential reasoning. In: Proceedings of the 7th Conference of the Cognitive Science Society, University of California, Irvine, CA, USA, pp 15–17

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

Pecorelli F, Di Nucci D, De Roover C, De Lucia A (2020a) A large empirical assessment of the role of data balancing in machine-learning-based code smell detection. Journal of Systems and Software p 110693

Pecorelli F, Palomba F, Khomh F, De Lucia A (2020b) Developer-driven code smell prioritization. In: Proceedings of the 17th International Conference on Mining Software Repositories, pp 220–231

Qing H, Biwen L, Beijun S, Xia Y (2015) Cross-project software defect prediction using feature-based transfer learning. In: Proceedings of the 7th Asia-Pacific Symposium on Internetware, pp 74–82

Quinlan JR (1987) Decision trees as probabilistic classifiers. In: Proceedings of the Fourth International Workshop on Machine Learning, Elsevier, pp 31–37

Ramirez A, Romero JR, 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,, IEEE, pp 223–232

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):1–44

Saidani I, Ouni A, Mkaouer MW (2020) Web service api anti-patterns detection as a multi-label learning problem. In: International Conference on Web Services, Springer, pp 114–132

Sangüesa R, Cabós J, Cortes U (1998) Possibilistic conditional independence: A similarity-based measure and its application to causal network learning. International Journal of Approximate Reasoning 18(1-2):145–167

dos Santos Neto BF, Ribeiro M, Da Silva VT, Braga C, De Lucena CJP, de Barros Costa E (2015) AutoRefactoring: A platform to build refactoring agents. Expert Systems with Applications 42:1652–1664

Sharma T (2019) Extending maintainability analysis beyond code smells. PhD thesis, University of Athens

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

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

Tsang S, Kao B, Yip KY, Ho WS, Lee SD (2009) Decision trees for uncertain data. IEEE transactions on knowledge and data engineering 23(1):64–78

Tsantalis N, Chatzigeorgiou A (2009) Identification of Move Method Refactoring Opportunities. IEEE Transactions on Software Engineering 35(3):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, Oliveto R, Di Penta M, De Lucia A, Poshyvanyk D (2017) When and why your code starts to smell bad (and whether the smells go away). IEEE Transactions on Software Engineering 43(11):1063–1088

Van Rijsbergen C (1979) Information retrieval

Vargha A, Delaney HD (2000) A critique and improvement of the cl common language effect size statistics of mcgraw and wong. Journal of Educational and Behavioral Statistics 25(2):101–132

Vassallo C, Grano G, Palomba F, Gall HC, 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 YG (2009) Tracking design smells: Lessons from a study of god classes. In: Proceedings of the 16th Working Conference on Reverse Engineering,, IEEE, pp 145–154

Whittle J, Sawyer P, Bencomo N, Cheng BH, Bruel JM (2009) Relax: Incorporating uncertainty into the specification of self-adaptive systems. In: Proceedings of the 17th International Requirements Engineering Conference, IEEE, pp 79–88

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

Witten IH, Frank E, Hall MA (2005) Data Mining: Practical Machine Learning Tools and Techniques. Morgan Kaufmann publishers

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

Zadeh LA (1978) Fuzzy sets as a basis for a theory of possibility. Fuzzy sets and systems 1(1):3–28

Zhu Z, Li Y, Tong H, Wang Y (2020) Cooba: Cross-project bug localization via adversarial transfer learning. In: Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI,

pp 3565–3571
Zimmermann T, Nagappan N, Gall H, Giger E, Murphy B (2009) Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In: Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, pp 91–100

Appendix A. Description of the handled code smells

In this study, we tested our approach on eight different types of code smells presented by Table 12. The considered anti-patterns are among the most considered code smells within the field of software maintenance (Fowler and Beck 1999), (Martin 2002), (Wirfs-Brock and McKean 2003), (Lanza and Marinescu 2007b), (Fontana et al. 2012), (Ouni 2014):

Table 12 List of the most handled code smells for the detection task in the literature(Boutaib et al. 2020).

Code Smell / Antipattern	**Description**
God Class (aka Blob)	It occurs when a class centralizes an important part of the system behavior while the remaining classes mainly contain data.
Data Class	It happens in the case where a class retains only data and not the complex functionalities.
Feature Envy	It arises when a method invokes much more methods from an external class than the invoked methods from the same englobing class.
Long Method	This smell refers to a method that is broad in terms of lines of code.
Duplicate code	Such smell arises when there are many redundancies for the same code fragment in many classes.
Long Parameter List	This smell happens when a method has a high number of parameter list.
Spaghetti Code	Such smell appears when the code structure is becoming much more complex and tangled.
Functional Decomposition	It arises when a class is developed in order to carry out a single function. This smell type is caused by the lack of OO developers experiences.

Appendix B. Description of the used metrics

Table 13 shows the list of the metrics that have been employed by GP (Ouni et al. 2013), BLOP (Sahin et al. 2014), and MOGP (Mansoor et al. 2017). To ensure a fair comparison, we have used all the listed metrics in Table 13.

Table 13 List of the considered measures (Boutaib et al. 2020).

Metric	Description
ANA - Average Number of Ancestors	Such measure corresponds to the average number of classes from which any class inherits information.
AOFD - Access Of Foreign Data	This measure is used to count the foreign attributes' number from unconnected classes, which are directly accessed or called through accessors (i.e., getters methods).
CAM - Cohesion Among Methods of Class	It is used to compute the relatedness between methods of the class according to the methods' parameter list.
CBO - Coupling Between Objects	This measure counts the number of classes, which call a function or access to a specific variable of a given class.
CIS - Class Interface Size	Such measure counts the number of existing methods that are public within a class. It is interpreted as the average of the total class in a design.
CM - Changing Method	This measure counts the number of distinct methods, which invoke the measure method.
DAM - Data Access Metric	It returns the ratio of the number of private (or protected) attributes to the total number of attributes mentioned in the class.
DCC - Direct Class Coupling	This measure returns the number of classes, in which a class is directly linked to. The metric considers the classes that are directly linked by the attribute declarations and message passing (refers to parameters) within methods.
DSC - Design Size in Classes	It is employed for counting the total number of classes within the design excepting the library classes that are imported.
LOC - Lines of Code	It returns the program size in terms of the instructions number within a class or method.
MFA - Measure of Functional Abstraction	This measure returns the ratio of the number of methods inherited by a class to the whole number of methods attainable by the 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	This measure returns the attributes' number regarding a given class within the program.
NOAM - Number of Accessor Methods	This metric is used to count the number of getters and setters that pertain to a class.
NOF - Number of Fields	This metric returns the classes fields' number.
NOH - Number of Hierarchies	This measure is employed to count the entire number of class hierarchies within the design.
NOM - Number of Methods	It counts the number of methods belonging to a given class.
NOPA - Number of Public Attributes	Such metric is used to count the public attributes' number belonging to a given class within the program.
NPA - Number of Private Attributes	This measure returns the private attributes number for a given class.
TCC - Tight Class Cohesion	This metric is employed to count the relative number of methods pairs of a class, which access to at least one attribute of the measured class.
WMC - Weighted Methods per Class	It computes the sum of the statistical complexity of a class based on the number of existing methods within the given class.
WOC - Weighted Of Class	This metric returns the number of functional methods in a given class divided by the entire number of members of the interface.

Appendix C. Basic concepts of possibility theory

1. The inconsistency metric

The inconsistency metric comes to measure the amount of conflict between the two opinions. More formally, we can calculate such degree by $Inc(\pi_1 \wedge \pi_2)$ where the conjunction $\wedge$ refers to the minimum (min) operator:

$$Inc(\pi_1, \pi_2) = Inc(\pi_1 \wedge \pi_2) = 1 - max_{\omega_i \in \Omega}\{min_{\omega_i \in \Omega}\{\pi_1(\omega_i), \pi_2(\omega_i)\}\} \quad (17)$$

2. The extreme knowledge forms of possibility theory

In possibility theory, we can differentiate the following two extreme knowledge forms of possibility distributions:

- Complete knowledge: $\exists\ \omega_k \in \Omega$, $\pi(\omega_k) = 1$ and all remaining states ω: $\pi(\omega) = 0$. In this case, there is only one fully possible element and the remaining ones are impossible. For example, in the case of identification, the complete knowledge occurs when a possibility degree of one smell type is equal to 1 and the possibility degrees of the other smell types are equal to 0 ($\pi(Blob)$=0, $\pi(Data\ Class)$=0, $\pi(Feature\ Envy)$=0, $\pi(Spaghetti\ Code)$=0, $\pi(Functional\ Decomposition)$=1, $\pi(Long\ Method)$=0, $\pi(Long\ Parameter\ List)$=0).
- Total ignorance:$\pi(\omega_k)$=1, $\forall\ \omega_k \in \Omega$ (i.e., all states ω_k are completely possible). In this case, all the elements in Ω are equally possible. In reality, the software engineer's opinion sometimes shows ignorance regarding the smelliness of a software class, and hence the possibility distribution is $\pi(Smelly)$=1, $\pi(Non-smelly)$=1.

3. Conjunctive and Disjunctive fusion modes and operators in possibility theory

The conjunctive fusion: This mode is employed when all the information sources are in agreement. This mode was defined by Dubois and Prade (2000) and represented by the following formula as follows:

$$\forall \omega \in \Omega,\ \ \pi_\wedge(\omega) = \otimes_{j=1\ldots n}\pi_j(\omega) \quad (18)$$

where $\otimes$ represents a [0,1]-valued operation specified on [0,1]$\times$[0,1], $\pi_\wedge$ refers to the conjunctive fusion mode of the possibility distribution π.

The disjunctive fusion: This mode of combination is defined when the information sources are mainly conflicting. The disjunctive fusion mode was proposed by Dubois and Prade (2000) and represented formally as follows:

$$\forall \omega \in \Omega,\ \ \pi_\vee(\omega) = \oplus_{j=1\ldots n}\pi_j(\omega) \quad (19)$$

where $\oplus$ represents a [0, 1]-valued operation specified on [0, 1] $\times$ [0, 1], and $\pi_\vee$ refers to the disjunctive fusion mode of the possibility distribution π.

Various candidates relative to the conjunctive fusion operators, named triangular norms (t-norms) (Klement et al. 2000), are employed. The most used ones are the following:

$$\begin{cases} Minimum: & \pi_1 \otimes \pi_2 = min(\pi_1, \pi_2) \\ Product: & \pi_1 \otimes \pi_2 = \pi_1 * \pi_2 \\ Lukasievicz\ \ t-norm: & \pi_1 \otimes \pi_2 = max(0, \pi_1 + \pi_2 - 1) \end{cases}$$

Assuming that two experts (E_1 and E_2) provide two opinions (π_1 and π_2) for the case of detection as follows:

$$\begin{matrix} \pi_1(Smelly) = 1 \\ \pi_1(Non-smelly) = 0.4 \end{matrix} \quad \text{And} \quad \begin{matrix} \pi_2(Smelly) = 1 \\ \pi_2(Non-smelly) = 0.23 \end{matrix}$$

One can notice that the two opinions are in agreement as they agree about the smelliness of the given software class (possibility degree equals to 1 over smelly class label in both opinions). The calculation of the conjunctive fusion operators is performed as follows:

- $Minimum: min(\pi_1, \pi_2) = min([1\ 0.4], [1\ 0.23]) = [1\ 0.23]$
- $Product: \pi_1 * \pi_2 = [1\ 0.4] * [1\ 0.23] = [1\ 0.1]$
- $Lukasievicz\ t-norm: max(0, \pi_1 + \pi_2 - 1) = max([0\ 0], [1\ 0.4] + [1\ 0.23] - [1\ 1]) = [1\ 0]$

The different disjunctive fusion operators are called the triangular conorms operation (t-conorms) (Klement et al. 2000). The duality relation leads to the following operators:

$$\begin{cases} Max: & \forall \omega \in \Omega,\ \ \pi_\vee(\omega) = max(\pi_1(\omega), \pi_2(\omega)) \\ Probabilistic: & \forall \omega \in \Omega,\ \ \pi_\vee(\omega) = \pi_1(\omega) + \pi_2(\omega) - \pi_1(\omega) * \pi_2(\omega) \\ Lukasievicz\ \ t-conorm: & \forall \omega \in \Omega,\ \ \pi_\vee(\omega) = min(1, \pi_1(\omega) + \pi_2(\omega)) \end{cases}$$

Supposing we have two experts (E_1 and E_2) that provide two opinions (π_1 and π_2) in disagreement for the case of detection as follows:

$$\begin{matrix} \pi_1(Smelly) = 0.4 \\ \pi_1(Non-smelly) = 1 \end{matrix} \quad \text{And} \quad \begin{matrix} \pi_2(Smelly) = 1 \\ \pi_2(Non-smelly) = 0.23 \end{matrix}$$

One can observe that the two opinions are in disagreement as they disagree about the smelliness of the passed software class. The calculation of the disjunctive fusion operators is performed as follows:

- $Max: max(\pi_1, \pi_2) = max([0.4\ 1], [1\ 0.23]) = [1\ 1]$
- $Probabilistic: \pi_1 + \pi_2 - \pi_1 * \pi_2 = [0.4\ 1] + [1\ 0.23] - [0.4\ 1] * [1\ 0.23] = [0.4\ 1] + [1\ 0.23] - [0.4\ 0.23] = [1\ 1]$

3. The calculation of the Affinity distance

In order to compute the affinity distance, we need to calculate the Manhattan distance that is expressed as follows:

$$d(\pi_1, \pi_2) = \frac{\sum_{i=1}^{n} |\pi_1(\omega_i) - \pi_2(\omega_i)|}{n} \tag{20}$$

To better clarify the calculation process of the information Affinity measure, we take as an example the two possibility distributions $\pi_1(0.4, 1)$ and $\pi_2(1, 0.23)$that represent the opinions of two experts for the case of detection. Therefore, the calculation of the Affinity is performed as follows:

- $d(\pi_1, \pi_2) = \frac{\sum_{i=1}^{n} |\pi_1(\omega_i) - \pi_2(\omega_i)|}{n} = \frac{|0.4-1|+|1-0.23|}{2} = 0.685$
- $Inc(\pi_1, \pi_2) = max_{\omega_i \in \Omega}(min_{\omega_i \in \Omega}(\pi_1(\omega_i), \pi_2(\omega_i))) = 1 - max(min(0.4, 1), min(1, 0.23)) = 1 - max(0.4, 0.23) = 1 - 0.4 = 0.6$
- $Aff(\pi_1, \pi_2) = 1 - \frac{\kappa * d(\pi_1, \pi_2) + \lambda * Inc(\pi_1, \pi_2)}{\kappa + \lambda} = 1 - \frac{0.5 * 0.685 + 0.5 * 0.6}{0.5 + 0.5} = 0.3575$

Appendix D. Demonstrating that *PF-measure_dist* and IAC exactly correspond to the *F-measure* and the accuracy in the certain case, respectively

This appendix is devoted to present examples illustrating the correspondence between the uncertain measures (i.e., *PF-measure_dist* and IAC) and the certain ones (i.e., *F-measure* and Accuracy).

1. ***Proof of correspondence between the PF-measure_dist and the F-measure:***
   A certain PBE includes only one fully possible class label (i.e., its possibility degree is equal to 1) and the remaining class labels are impossible (i.e., their possibility degrees are equal to 0). In this appendix, we demonstrate that the *PF-measure_dist* working process corresponds to the conventional *F-measure* one in a certain environment. Figure 19 presents two PBEs (A) and (B), where PBE (A) is the ground truth and PBE (B) is the predicted one. It is important to note that the two PBEs (A) and (B) correspond to the detection process and their labels have the form of possibility distributions. The *PF-measure_dist* is calculated as follows. First, we start by measuring the closeness between the initial instance (from the ground truth) and the predicted one using sd($\vec{I_j}$) (cf. Eq. 10). Then, we normalize the obtained sd($\vec{I_j}$) value using NSD($\vec{I_j}$) (cf. Eq. 11). Finally, based on the comparison between the predicted obtained class labels and the initial ones, we add the obtained NSD($\vec{I_j}$) value to one of these measures: TP_dist (if the actual Smelly classes correctly classified) or FP_dist (if the actual Smelly classes miss-classified as Non-smelly) or TN_dist (if the actual Non-smelly classes correctly classified) or FN_dist (Actual Non-smelly classes miss-classified as Smelly). One can see from Figure 19 that the possibility distribution value of $Class_1$ in PBE (A) is similar to the predicted one in PBE (B). The sd($\vec{I_j}$) and NSD($\vec{I_j}$) values of $Class_1$ are calculated as follows:
   $sd(\vec{I_1}) = (0-0)^2 + (1-1) = 0$ and $NSD(\vec{I_1}) = 1$.
   TP_dist= NSD($\vec{I_1}$) +TP_dist, TN_dist= NSD($\vec{I_1}$) +TN_dist, FP_dist= NSD($\vec{I_1}$)

+FP_dist, FN_dist= NSD($\vec{I_1}$) +FN_dist
$\Longrightarrow$ TP_dist=0, TN_dist=1, FP_dist=0, FN_dist=0.

Similarly, sd($\vec{I_6}$) and NSD($\vec{I_6}$) values of $Class_6$ are calculated as follows:
$sd(\vec{I_6}) = (1-0)^2 + (0-1) = 2$ and $NSD(\vec{I_6}) = 1$
$\Longrightarrow$ TP_dist=0, TN_dist=0, FP_dist=0, FN_dist=1.
TP_dist= NSD($\vec{I_6}$) +TP_dist, TN_dist= NSD($\vec{I_6}$) +TN_dist, FP_dist= NSD($\vec{I_6}$) +FP_dist, FN_dist= NSD($\vec{I_6}$) +FN_dist

This process is continued until reaching the final instance. Thus, we will obtain the following values:
$\Longrightarrow$ TP_dist=2, TN_dist=3, FP_dist=0, FN_dist=1.
Based on the obtained values of TP_dist, TN_dist, FP_dist, FN_dist (which are equal to 2, 5, 1, and 2, respectively), we calculate the Precision_dist and Recall_dist values (which are equal to 0.6667 and 0, respectively). These latter are used to compute the *PF-measure_dist* (*PF-measure_dist*=0.571). From this example, one can observe that the calculated *PF-measure_dist* value is equal to the *F-measure* one.

2. ***Proof of correspondence between the IAC and the Accuracy:***
The IAC measure (cf. Eq. 16) is based on the distance d (cf. Eq. 20) and the Inconsistency Inc (cf. Eq. 17) and it is calculated as follows:

$$d(\pi_1^{ini}, \pi_1^{pred}) = \frac{|0-0| + |1-1|}{2} = 0$$

$$Inc(\pi_1^{ini}, \pi_1^{pred}) = 1 - max(min(0,0), min(1,1)) = 0$$

$$Aff(\pi_1^{ini}, \pi_1^{pred}) = 1 - \frac{0.5*0 + 0.5*0}{1} = 1,$$

where the two parameters κ and λ are set to 0.5.
Similarly, d and Inc values of $Class_6$ are calculated as follows:

$$d(\pi_6^{ini}, \pi_6^{pred}) = \frac{|0-1| + |1-0|}{2} = 1$$

$$Inc(\pi_6^{ini}, \pi_6^{pred}) = 1 - max(min(0,1), min(1,0)) = 1$$

$$Aff(\pi_6^{ini}, \pi_6^{pred}) = 1 - \frac{0.5*1 + 0.5*1}{1} = 0,$$

$$d(\pi_6^{ini}, \pi_6^{pred}) = \frac{|0-0| + |1-1|}{2} = 0$$

$$Inc(\pi_6^{ini}, \pi_6^{pred}) = 1 - max(min(0,0), min(1,1)) = 0$$

$$Aff(\pi_6^{ini}, \pi_6^{pred}) = 1 - \frac{0.5*0 + 0.5*0}{1} = 1$$

This process is continued until reaching the final instance. Thus, we will obtain the following values:

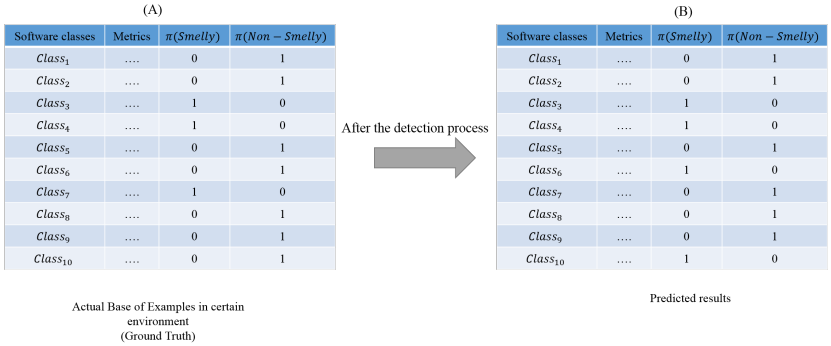


Fig. 19 Example of the obtained PBE after the detection process

$$Aff(\pi_1^{ini},\pi_1^{pred}) = 1, \quad Aff(\pi_2^{ini},\pi_2^{pred}) = 1, \quad Aff(\pi_3^{ini},\pi_3^{pred}) = 1,$$
$$Aff(\pi_4^{ini},\pi_4^{pred}) = 1, \quad Aff(\pi_5^{ini},\pi_5^{pred}) = 1, \quad Aff(\pi_6^{ini},\pi_6^{pred}) = 0,$$
$$Aff(\pi_7^{ini},\pi_7^{pred}) = 0, \quad Aff(\pi_8^{ini},\pi_8^{pred}) = 1, \quad Aff(\pi_9^{ini},\pi_9^{pred}) = 1,$$
$$Aff(\pi_{10}^{ini},\pi_{10}^{pred}) = 0.$$
$$\Longrightarrow IAC = \frac{1}{n} \times \sum_{i=1}^{10} Aff(\pi_i^{ini},\pi_i^{pred}) = 0.7$$

Based on the obtained values of the TP and TN (which are 2 and 5, respectively), the Accuracy value is calculated as follows:

$$Accuracy = \frac{TP+TN}{Totalnumberofinstances} = \frac{2+5}{10} = 0.7$$

Based on this example, one can conclude that the IAC measure corresponds to the Accuracy measure in a certain environment.

Appendix E: Comparison between ADIPE and the remaining approaches using the *AUC* and *AURPC* measures

In this appendix, we present the obtained results of ADIPE, DECOR, GP, MOGP, and BLOP for the detection and identification cases under a certain environment using the *AUC* (Area Under ROC Curve) metric. In order to ensure a fair comparison, we evaluate the performance of the five compared algorithms in the certain environment since DECOR, GP, MOGP, and BLOP are not conceived to deal with uncertainty. Tables 14 and 15 show the *AUC* values of the different considered approaches including ADIPE for the detection and identification tasks, respectively. Based on Table 14, one can see that ADIPE surpasses all the considered approaches with an *AUC* value that lies within [0.9225, 0.9452], while BLOP succeeds to obtain the second-best performance with an *AUC* value that ranges between 0.1923 and 0.3256. DECOR, GP, and MOGP have a lower performance since their obtained *AUC* values are equal to 0.114, 0.2382, and 0.2874, respectively. The outperformance of ADIPE could be explained by the fact that its fitness function (i.e., *PF-measure_dist*) is insensitive to the problem of imbalanced data. GP, MOGP, and BLOP have a lower performance since their fitness functions are not suitable to deal with imbalanced data, which may create ineffective detection rules. DECOR has the lowest *AUC* values since it is a rule-based algorithm

Table 14 *AUC* median values of ADIPE, DECOR, GP, MOGP, and BLOP for 31 runs of the detection task at the certain environment.

Software projects	ADIPE	DECOR	GP	MOGP	BLOP	
	0.9263	0.114	0.2382	0.2874		
GanttProject	**(+ + + +)**	(+ + +)	(- +)	(-)		0.3256
	(l l l l)	(m m m)	(s m)	(s)		
	0.9314	0.1532	0.1756	0.2396		
ArgoUML	**(+ + + +)**	(- + +)	(- +)	(-)		0.3174
	(l l l l)	(s m m)	(s m)	(s)		
	0.9225	0.0986	0.2069	0.2815		
Xercess-J	**(+ + + +)**	(+ + +)	(- -)	(-)		0.3082
	(l l l l)	(m m l)	(s s)	(s)		
	0.9452	0.0947	0.1173	0.2061		
Jfreechart	**(+ + + +)**	(- + +)	(+ +)	(-)		0.2257
	(l l l l)	(s m l)	(m m)	(s)		
	0.9418	0.0803	0.106	0.1843		
Azureus	**(+ + + +)**	(- + +)	(- +)	(-)		0.2108
	(l l l l)	(s m m)	(s m)	(s)		
	0.9417	0.0412	0.0745	0.1388		
ApacheAnt	**(+ + + +)**	(- + +)	(- +)	(-)		0.1923
	(l l l l)	(s m m)	(s m)	(s)		

- The sign "+" at the i^{th} position means that the algorithm *AURPC* median value is statically different from the i^{th} algorithm value. The sign "-" means the opposite.
- The effect sizes values (small (s), medium (m), and large (l)) using the A-statistics are given.
- Best *AURPC* values are highlighted in Bold. Second-best *AURPC* values are underlined.

characterized with a set of manually defined detection rules. Table 15 shows similar results for the identification case. The *AUC* values of ADIPE are slightly degraded, while the *AUC* values of DECOR, GP, MOGP, and BLOP are significantly degraded due to the high imbalance ratio over the identification process. It should be noted that the A-statistic results are similar to the obtained ones in Tables 6 and 8. Similar results are obtained for the *AURPC* (Area Under Recall Precision Curve) metric based on Tables 16 and 17. One can observe that ADIPE has the best performance in terms of *AURPC* in all the considered software projects, while BLOP has the second best performance. This observation clearly demonstrates the ability of our approach to deal with imbalanced data.

Table 15 *AUC* median values of ADIPE, DECOR, GP, MOGP, and BLOP for 31 runs of the identification task at the certain environment.

Code Smell	ADIPE	DECOR	GP	MOGP	BLOP
Blob	**0.9306** **(+ + + +)** **(l l l l)**	0.3425 (- + +) (s m m)	0.3342 (+ +) (m m)	0.4178 (-) (s)	0.4389
Data Class	**0.9114** **(+ + +)** **(l l l)**	N/A	0.2605 (+ +) (m m)	0.3294 (-) (s)	0.3762
Feature Envy	**0.9022** **(+ + +)** **(l l l)**	N/A	0.2581 (+ +) (m m)	0.3206 (-) (s)	0.3711
Long Method	**0.8876** **(+ + +)** **(l l l)**	N/A	0.2411 (+ +) (m m)	0.308 (-) (s)	0.3419
Duplicate Code	**0.8734** **(+ + +)** **(l l l)**	N/A	0.1975 (+ +) (m l)	0.2766 (+) (m)	0.3274
Long Parameter List	**0.8875** **(+ + +)** **(l l l)**	N/A	0.141 (+ +) (l l)	0.2392 (+) (m)	0.2936
Spaghetti Code	**0.8761** **(+ + + +)** **(l l l l)**	0.0773 (- + +) (s l l)	0.1197 (+ +) (m l)	0.1854 (+) (m)	0.2553
Functional Decomposition	**0.8569** **(+ + + +)** **(l l l l)**	0.0619 (- + +) (s l l)	0.1083 (+ +) (m l)	0.1563 (+) (m)	0.1998

- The sign "+" at the i^{th} position means that the algorithm *AUC* median value is statically different from the i^{th} algorithm value. The sign "-" means the opposite.
- The effect sizes values (small (s), medium (m), and large (l)) using the A-statistics are given.
- Best *AUC* values are highlighted in Bold. Second-best *AUC* values are underlined.

Appendix F: Motivations for the Evolutionary design of PDTs.

In this appendix, we highlight the advantages of using EAs in optimizing PDTs. In the literature, machine learning algorithms have been widely used in designing PDTs (Barros et al. 2012; Al-Sahaf et al. 2019). However, existing machine learning algorithms could lead the solution to got stack into local optima. One of the well-known EAs is the GA that has proven its ability to escape from local optima (Holland 1992). Figure 20 illustrates the schema for PDT configuration of two local optima as well as one global optimum search using the $PF - measure\_dist$. Each point on the x-axis represents a PDT configuration for ease of visualization, while the y-axis displays the value of the $PF - measure\_dist$ (configuration quality). According to this schema, starting with solution (configuration) A, the greedy algorithm will converge to one of the two globally optimal PDT structures G_1 or G_2. Likewise, if it begins induction with solution B, it could approximate the global optimum G_3 or the local one G_2. As a result, the chance that a greedy PDT induction method finding a closer globally-optimal configuration is extremely low, whereas GAs do not have this problem due to two features. From one side, similarly to any EA approach, the GA has a global search capability because it can identify promising regions within the search space, more

Table 16 *AURPC* median values of ADIPE, DECOR, GP, MOGP, and BLOP for 31 runs of the detection task at the certain environment.

Software projects	ADIPE	DECOR	GP	MOGP	BLOP	
	0.9243	0.1109	0.2354	0.2822		
GanttProject	**(+ + + +)**	(+ + +)	(- +)	(-)		0.3152
	(l l l l)	(m m m)	(s m)	(s)		
	0.9302	0.1513	0.1721	0.2349		
ArgoUML	**(+ + + +)**	(- + +)	(- +)	(-)		0.3106
	(l l l l)	(s m m)	(s m)	(s)		
	0.9186	0.0963	0.2011	0.2783		
Xercess-J	**(+ + + +)**	(+ + +)	(- -)	(-)		0.3051
	(l l l l)	(m m l)	(s s)	(s)		
	0.9430	0.0920	0.1133	0.2046		
Jfreechart	**(+ + + +)**	(- + +)	(+ +)	(-)		0.2239
	(l l l l)	(s m l)	(m m)	(s)		
	0.9410	0.0796	0.1019	0.1827		
Azureus	**(+ + + +)**	(- + +)	(- +)	(-)		0.2091
	(l l l l)	(s m m)	(s m)	(s)		
	0.9409	0.0403	0.0715	0.1367		
ApacheAnt	**(+ + + +)**	(- + +)	(- +)	(-)		0.1910
	(l l l l)	(s m m)	(s m)	(s)		

- The sign "+" at the i^{th} position means that the algorithm *AURPC* median value is statically different from the i^{th} algorithm value. The sign "-" means the opposite.
- The effect sizes values (small (s), medium (m), and large (l)) using the A-statistics are given.
- Best *AURPC* values are highlighted in Bold. Second-best *AURPC* values are underlined.

precisely; regions close to G_1, G_2, and G_3. This could be done by simultaneously evolving up an entire population (a number of configurations) rather than just a single configuration. On the other side, the binary tournament operator allows the acceptance of poor configurations ($PF - measure\_dist$ degradation). Based on this fact, the GA can avoid falling into local optima like solutions close to G_1 and G_2. Such operator carries out (N/2) iterations to pick up (N/2) offsprings for the reproduction process. Accordingly, when two PDT configurations are chosen from the mating pool as parents for crossover, these parents could include both good and bad PDT configurations. As a result, the GA permits the fitness function to deteriorate, allowing it to: (1) avoid local optima like solution G_2 and then (2) guide the search process through the globally optimal PDT configuration G_3. Another reason for the employment of the GA algorithm is that it has shown good performance in the certain environment (Boutaib et al. 2020). Motivated by the mentioned advantages of the GA, we have proposed to assess its performance over the uncertain environment.

Table 17 *AURPC* median values of ADIPE, DECOR, GP, MOGP, and BLOP for 31 runs of the identification task at the certain environment.

Code Smell	ADIPE	DECOR	GP	MOGP	BLOP
Blob	**0.9285** **(+ + + +)** **(l l l l)**	0.3307 (- + +) (s m m)	0.3326 (+ +) (m m)	0.4168 (-) (s)	0.4359
Data Class	**0.9109** **(+ + +)** **(l l l)**	N/A	0.2583 (+ +) (m m)	0.3274 (-) (s)	0.3722
Feature Envy	**0.8995** **(+ + +)** **(l l l)**	N/A	0.2553 (+ +) (m m)	0.3186 (-) (s)	0.3690
Long Method	**0.8865** **(+ + +)** **(l l l)**	N/A	0.2396 (+ +) (m m)	0.2993 (-) (s)	0.3405
Duplicate Code	**0.8712** **(+ + +)** **(l l l)**	N/A	0.1962 (+ +) (m l)	0.2751 (+) (m)	0.3259
Long Parameter List	**0.8868** **(+ + +)** **(l l l)**	N/A	0.1392 (+ +) (l l)	0.2376 (+) (m)	0.2912
Spaghetti Code	**0.8753** **(+ + + +)** **(l l l l)**	0.0761 (- + +) (s l l)	0.1183 (+ +) (m l)	0.1839 (+) (m)	0.2521
Functional Decomposition	**0.8543** **(+ + + +)** **(l l l l)**	0.0603 (- + +) (s l l)	0.1056 (+ +) (m l)	0.1544 (+) (m)	0.1965

- The sign "+" at the i^{th} position means that the algorithm *AURPC* median value is statically different from the i^{th} algorithm value. The sign "-" means the opposite.
- The effect sizes values (small (s), medium (m), and large (l)) using the A-statistics are given.
- Best *AURPC* values are highlighted in Bold. Second-best *AURPC* values are underlined.

Appendix G: Comparison based on the CPU time between the peer algorithms

In this Appendix, we compare the considered algorithms in our experimental study (i.e., GP, MOGP, BLOP, and ADIPE) from the CPU time viewpoint. In fact, all the algorithms under comparison were executed on machines with Intel Core i7 2.20 GHz processor and 8 GB RAM. We also note that we have used the simplest pseudo-parallel approach, which is the multi-threading. Table 18 shows the CPU times obtained in each software project for each induction algorithm (i.e., GP, MOGP, BLOP, and ADIPE). For fairness of comparison, we use the same number of evaluation for all the considered algorithms. This stopping criterion is set to 256,500 for all the used software projects. One can see from Table 18 that the CPU time consumed by ADIPE is higher than the ones of GP, MOGP, and BLOP since the update procedure of the reproduction operators, the building structure of our PDT, and the evaluation process are a little time consuming. Table 19 reports the CPU times obtained by PDT ensemble, DECOR, GP-Tree, MOGP-Tree, and BLOP-Tree on the unseen project (i.e., Lucene v1.4.1[24] with 154 software classes and 41 smells). One can observe from Table 19 that the

[24]http://lucene.apache.org/

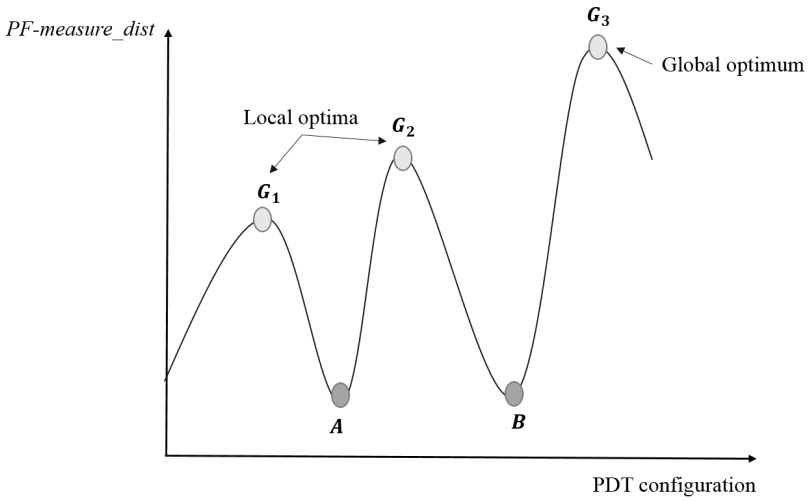


Fig. 20 Illustration of the schema for PDT configuration of two local optima as well as one global optimum search using the $PF - measure\_dist$

Table 18 CPU time consumed in hours by GP, MOGP, BLOP, and ADIPE on the six considered software projects (i.e., GanttProject, ArgoUML, Xercess-J, JFreeChart, Apache Ant, Azureus).

Approaches	CPU time
GP	3h 17 min
MOGP	6h 41 min
BLOP	7h 29 min
ADIPE	8h 11 min

Table 19 CPU time consumed in milliseconds by DECOR, GP-Tree, MOGP-Tree, BLOP-Tree, and PDT ensemble on the unseen project Lucene v1.4.1

Approaches	CPU time (ms)
DECOR	10
GP-Tree	18
MOGP-Tree	23
BLOP-Tree	25
PDT ensemble	50

CPU time of ADIPE is slightly higher than the ones of its competitors since it uses a PDT ensemble instead of a single detector, which is the case of the other approaches.

Appendix H: Investigation of ADIPE performance on a cross-project

In this appendix, we show how the proposed approach is able to achieve good performance on cross-projects code smell detection. As shown in Figure 21, ADIPE is trained based on the source projects that have been used to construct the PBE. In an industrial context, these software projects are labeled by software engineers that may have some uncertainty and doubtfulness regarding the smelliness of a number of software classes. The output of ADIPE is a PDT ensemble that are applied to the target project that is not considered in the training process. It is important to note that before the application of the PDT ensemble, a TCA (Transfer Component Analysis) (Pan et al. 2010) method is used as a Domain Adaptation module. This latter aims to have similar distributions between the source projects and the target one, since having different distributions between the source and target projects may have a negative influence on the performance of the classifiers. After the application of the Domain Adaptation module, the PDTs could be applied to predict the labels of the existing smelly instances of the target project. Therefore, the uncertainty of the software engineers are transferred from the source projects to the target ones through the application of the PDT detectors (PDT ensemble). In the validation step, the software engineers label the software classes of the target project to be able to compare it with the predicted software class labels obtained by the PDT detectors. It is worth noting that the labeling step is performed by the same software engineers that have constructed the BE of the source projects. An important threat that should not be ignored is that the validation of the results must be performed by the developers who developed the Lucene software project and not those who built the base of examples. Hence, since the training step, the environment nature (certain or uncertain) of the source and target projects is specified based on the opinions of the software engineers. This validation process allows us to evaluate the effectiveness of the constructed PDTs in transferring the uncertainty of the human experts when introducing unseen software cross-projects. In this appendix, we have investigated the performance of our approach on the unseen Lucene 1.4.1 software project using the *PF-measure_dist* and IAC measures in the uncertain environment and the F-measure and the Accuracy in the certain one. Table 20 shows that ADIPE outperforms the remaining approaches (DECOR, GP, MOGP, and BLOP) in both environments with respect to all the considered metrics. ADIPE achieves 0.9115 and 0.9273 in terms of *PF-measure_dist* and IAC, respectively. The second best performance is obtained by BLOP where it succeeds to obtain 0.2183 and 0.2330 in terms of *PF-measure_dist* and IAC measures. The same performance is obtained by the compared algorithms in the certain environment.

Table 20 *PF-measure_dist* and IAC (F-measure and Accuracy) median scores of ADIPE, DECOR, GP, MOGP, and BLOP for 31 runs of the detection task at the uncertain (and certain) environments on the Lucene v1.4.1 unseen project.

Environment	ADIPE	DECOR	GP	MOGP	BLOP	
			PF-measure_dist			
Uncertain	**0.9115**	0.1267	0.1583	0.2019		
	(++++)	(- ++)	(+ +)	(-)		0.2183
	(l l l l)	(smm)	(m m)	(s)		
			IAC			
	0.9273	0.1349	0.1709	0.2254		
	(+ + + +)	(+ + +)	(+ +)	(-)		0.2330
	(l l l l)	(m m m)	(m m)	(s)		
			PF-measure_dist (F-measure)			
Certain	**0.9269**	0.1486	0.1671	0.2241		
	(+ + + +)	(- + +)	(+ +)	(-)		0.2392
	(l l l l)	(s m m)	(m m)	(s)		
			IAC (PCC)			
	0.9364	0.1658	0.1742	0.2369		
	(+ + + +)	(- + +)	(+ +)	(+)		0.2587
	(l l l l)	(s m m)	(m m)	(m)		

- The sign "+" at the i^{th} position means that the algorithm metric median value (*PF-measure_dist*) (abbreviated by PF_d) or *IAC*(Accuracy) median value) is statically different from the i^{th} algorithm value. The sign "-" means the opposite.
- The effect sizes values (small (s), medium (m), and large (l)) using the A-statistics are given.
- Best obtained metrics values are highlighted in Bold. Second-best obtained metrics values are underlined.

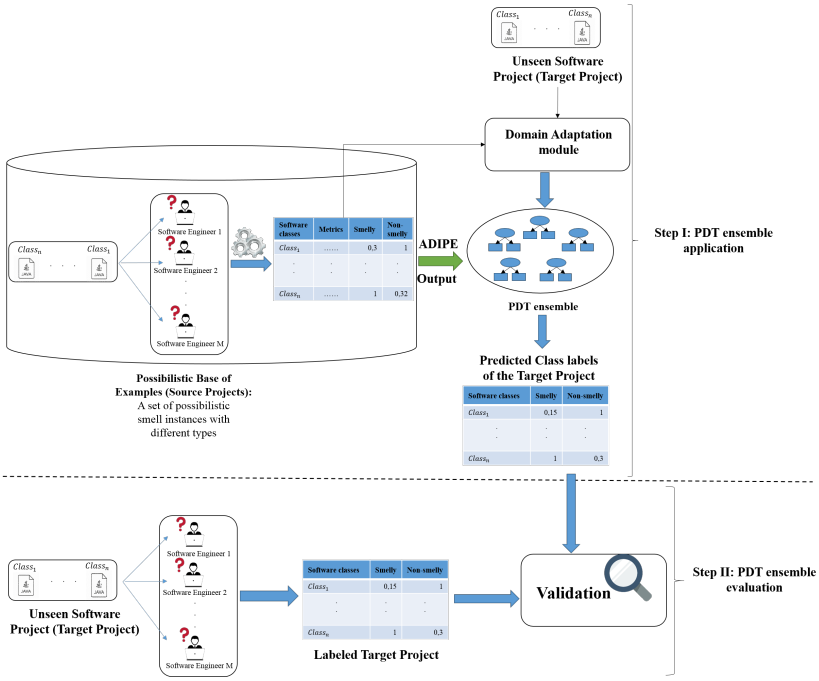


Fig. 21 Illustration of the application of the PDT ensemble on a cross-project problem.