Uncertainty-wise Software Anti-patterns Detection: A Possibilistic Evolutionary Machine Learning Approach

Sofien Boutaib[a,*], Maha Elarbi[a], Slim Bechikh[a,*], Carlos A. Coello Coello[b], Lamjed Ben Said[a]

[a]*SMART Lab, ISG, University of Tunis, Tunisia*
[b]*Departamento de Computación, CINVESTAV-IPN, Mexico City, Mexico*
Basque Center for Applied Mathematics (BCAM) & Ikerbasque

Abstract

Context. Code smells (a.k.a. anti-patterns) are manifestations of poor design solutions that can deteriorate software maintainability and evolution. ***Research gap.*** Existing works did not take into account the issue of uncertain class labels, which is an important inherent characteristic of the smells detection problem. More precisely, two human experts may have different degrees of uncertainty about the smelliness of a particular software class not only for the smell detection task but also for the smell type identification one. Unluckily, existing approaches usually reject and/or ignore uncertain data that correspond to software classes (i.e. dataset instances) with uncertain labels. Throwing away and/or disregarding the uncertainty factor could considerably degrade the detection/identification process effectiveness. From a solution approach viewpoint, there is no work in the literature that proposed a method that is able to detect and/or identify code smells while preserving the uncertainty aspect. ***Objective.*** The main goal of our research work is to handle the uncertainty factor, issued from human experts, in detecting and/or identifying code smells by proposing an evolutionary approach that is able to deal with anti-patterns classification with uncertain labels. ***Method.*** We suggest Bi-ADIPOK, as an

*Corresponding author
Email addresses: boutaibsofien@yahoo.fr (Sofien Boutaib), arbi.maha@yahoo.com (Maha Elarbi), slim.bechikh@fsegn.rnu.tn (Slim Bechikh), ccoello@cs.cinvestav.mx (Carlos A. Coello Coello), lamjed.bensaid@isg.rnu.tn (Lamjed Ben Said)

effective search-based tool that is capable to tackle the previously mentioned challenge for both detection and identification cases. The proposed method corresponds to an EA (Evolutionary Algorithm) that optimizes a set of detectors encoded as PK-NNs (Possibilistic K-nearest neighbors) based on a bi-level hierarchy, in which the upper level role consists on finding the optimal PK-NNs parameters, while the lower level one is to generate the PK-NNs. A newly fitness function has been proposed fitness function PomAURPC-OVA_dist (Possibilistic *modified Area Under Recall Precision Curve One-Versus-All*_distance, abbreviated PAURPC_d in this paper). Bi-ADIPOK is able to deal with label uncertainty using some concepts stemming from the Possibility Theory. Furthermore, the PomAURPC-OVA_dist is capable to process the uncertainty issue even with imbalanced data. We notice that Bi-ADIPOK is first built and then validated using a possibilistic base of smell examples that simulates and mimics the subjectivity of software engineers opinions. ***Results.*** The statistical analysis of the obtained results on a set of comparative experiments with respect to four relevant state-of-the-art methods shows the merits of our proposal. The obtained detection results demonstrate that, for the uncertain environment, the PomAURPC-OVA_dist of Bi-ADIPOK ranges between 0.902 and 0.932 and its IAC lies between 0.9108 and 0.9407, while for the certain environment, the PomAURPC-OVA_dist lies between 0.928 and 0.955 and the IAC ranges between 0.9477 and 0.9622. Similarly, the identification results, for the uncertain environment, indicate that the PomAURPC-OVA_dist of Bi-ADIPOK varies between 0.8576 and 0.9273 and its IAC is between 0.8693 and 0.9318. For the certain environment, the PomAURPC-OVA_dist lies between 0.8613 and 0.9351 and the IAC values are between 0.8672 and 0.9476. With uncertain data, Bi-ADIPOK can find 35% more code smells than the second best approach (i.e., BLOP). Furthermore, Bi-ADIPOK has succeeded to reduce the number of false alarms (i.e., misclassified smelly instances) by 12%. In addition, our proposed approach can identify 43% more smell types than BLOP and reduces the number of false alarms by 32%. The same results have been obtained for the certain environment, demonstrating Bi-ADIPOK's ability to deal with such environ-

ment.

Keywords: Code smells detection, Data uncertainty, Possibility theory, Evolutionary Algorithm, Possibilistic K-NN.

---

1. Introduction

Several studies on software engineering have demonstrated that the deadline pressure and the developer's inexperience are among the factors that may lead to the introduction of the so-called technical debt [1, 2], which refers to a group of design issues that may affect the software's maintenance as well as its evolution in the future [3, 4]. Code smells (a.k.a. Anti-patterns) [5] are among the code technical debt' indications, i.e., poor design solutions (or best practice violations) that developers introduce to meet the new requirements in the software system. Previous empirical studies have revealed that code smells are serious issues to maintainability and evolution of source code as they deteriorate the developer's ability to comprehend the software source code and render the classes that are affected, more change- and fault-prone [6, 7, 8, 9]. In fact, enhancing the code quality may lead to the appearance of major bugs in the new software system. Therefore, it is important to detect well the smelly parts of the software system before proceeding to the refactoring process that consists of modifying the internal code structure with keeping the external behavior of the system unchanged.

Motivated by this observation, researchers have paid attention to code smell detection [10] and they have proposed different techniques to automatically detect code smells within software systems codebases [4]. These techniques could be classified into three categories: (1) rule/heuristic-based approaches [11], (2) machine learning-based approaches [12], and (3) search-based ones [13]. Mantyla et al. have been discussing the uncertainty issue as one of the major individual human factors that may influence software engineers' decisions about the smelliness of software classes for more than fifteen years, and more specifically since 2004. These choices are affected by their experiences as well as intuitions. As a

result, developers may have differing viewpoints, owing to their varying levels of knowledge and expertise. A considerable number of existing techniques can be considered as rule/heuristic-based approaches as they proceed as follows. First, they calculate the values for the set of chosen metrics. Second, some thresholds are applied on the on considered metrics to distinguish smelly code fragments from non-smelly ones. These techniques have shown good performance, however their main drawback is the specification of the threshold parameter that pushes the practitioners to ignore a portion of occurrences of code smells [14, 15]. The threshold selection influences the code smell detectors performances. Different from the rule/heuristic-based techniques, machine learning-based techniques train a classifier model based on a base of examples. The existing approaches use classifiers to detect the smelly instances rather than employing predefined thresholds values on the calculated metrics. However, machine learning-based techniques provide locally-optimal classifiers since their model construction process is performed in a greedy way [16, 17]. Moreover, the existing techniques need a high-quality base of examples with sufficient sizes to build classifiers models [10]. The latter category, i.e., search-based one, overcome such issues, as they can tune the thresholds based on the base of examples (including good and bad designs) [18]. In addition, the search-based techniques optimize a set of code smell detectors (i.e., detection rules) using evolutionary algorithms, which aid to exploit (near) globally-optimal detectors [19, 20].

Most of the existing techniques (detectors), in particular those belonging to the two latter described categories, have displayed considerable detection performance, but they still present some critical limitations that minimize their adoption in the industry. Some studies [21, 22] mentioned that the obtained results from the automated detectors, regarding the smelliness of software classes, are interpreted in a subjective way due to the differentiation between developers in terms of knowledge and expertise. This could be explained by the doubtfulness of the software engineers interpretations concerning the smelliness of software classes components. Two software engineers could identify various smell types differently over a software class, as demonstrated in previous work

```
    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);
    }
}
```

Figure 1: A Long Method smell type called rtrim () method for the Apache Common CLI [24]. When using the JDedoroant advisor, this method is considered a Long method, whereas when using the PMD advisor, it is considered a normal method.

[23]. The fact that they may have varying levels of experience could explain their subjectivity and uncertainty. As a result, one software engineer may label a class Smelly (i.e., containing a smell type) and another software engineer may label it as Non-smelly. To simulate such situation, we picked two different advisors (JDeodorant and PMD) in the aim to mimic expert opinions for the detection of the Long Method smell type. Figure 1 shows a code fragment from the Apache Commons CLI, specifically a method called rtrim() [24]. The rtrim() method is a Long method smell according to the JDedoroant advisor, but it is a normal method according to the PMD. Such subjectivity is explained by the fact that each of those advisors uses different detection rules. Furthermore, human software engineers are sources of subjectivity because they can only express their uncertainty using likelihood values. Existing research has shown that likelihood values (i.e., probability theory) are ineffective when dealing with uncertain data [25]. To solve this problem, we propose using the possibility theory, which has been shown to be adequate in the case of uncertain data [26].

The idea to alleviate such a problem was inspired by the studies that are conducted in the data mining field. Thus, this problem can be represented as an uncertain class label problem [27]. Generally, the dependent variables refer to the smelliness of a class (i.e., presence/absence of smell), but in our case, these

variables are likelihood values that represent the doubtfulness of the developer towards the degree of membership of a class to one of the two categories (i.e., smelly/non-smelly). However, most of the existing works do not consider (often ignore) the uncertain data and replace it with a certain (crisp) one. Such practice is done a-posteriori, specifically when the code smell occurrences have been introduced to software developers: this does not allow them to profit from the highest information amount covered up in the data [26] and thus it might break down the overall performance of the smell detectors. Scientists of data mining and machine learning domains have suggested a multitude of uncertainty theories, in which possibility theory is one of them.

In this research paper, we present the first search-based tool in the SE domain and in particular the Search Based Software Engineering (SBSE) one, named Anti-pattern Detection and Identification using Possibilistic Optimized K-NNs (Bi-ADIPOK) to detect and identify code smells under uncertainty. Indeed, the detection process consists of separating whether a given class is affected by a generic code smell, while the identification process target calling attention to the presence of a specific sort of code smell (e.g., a Feature Envy).

Motivated by the remarkable performance of the K-Nearest Neighbors (K-NN) under Possibilistic Framework (abbreviated Possibilistic K-NN (PK-NN) in this paper) [28] classifier, which is a blend between Possibility theory and K-NN, Bi-ADIPOK evolves a set of PK-NNs using the GA (Genetic Algorithm) metaheuristic. It requires as input a possibilistic base of examples (PBE) characterized with uncertain class labels that are designed in the form of possibility distributions and generate as output a set of optimized possibilistic detectors (i.e., optimized PK-NNs). To verify the smelliness of a class, Bi-ADIPOK analyzes the generated PK-NNs and merges the resulting possibility distributions to obtain a single one that indicates the presence of a peculiar smell. Similarly, in the identification problem, Bi-ADIPOK launches the GA upon a base of examples including a single smell type and then the same task for the fusion of the distribution is carried out.

Given that Bi-ADIPOK adopts numerous techniques from the computational

intelligence field, it is important to justify our choices:

1. The interesting performance of PK-NNs in uncertain data classification is well-appreciated through the the machine learning literature thanks to their capability to learn from a training set in which uncertain class labels are represented by possibility distributions [28];
2. The capability of GA to escape local optima in the PK-NN search space; which is not the case of state-of-the art greedy machine learning algorithms for PK-NN induction;
3. The well-suited structuring of PK-NNs as uncertain classifiers, which is not the case of search-based detection methods that usually optimize a set of ad-hoc rules; and
4. The adaptive fusion of possibilistic distributions through conjunctive and disjunctive aggregations; which eases the decision making whatever are the states of the code smell detectors (in a concordance or disagreement [29]).

Our Bi-ADIPOK approach performance is assessed on the basis of a detailed empirical study including six well-known open-source software systems inundated by uncertainty. The comparisons are conducted regarding four relevant state-of-the-art approaches. Our experimental results demonstrate that Bi-ADIPOK has better performance on both tasks: detection and identification. In summary, the main contributions of this paper are:

1. Constructing a new base of possibilistic smell instances in view of emulating the subjective and uncertain opinions of the software developers where the uncertain and subjective opinions are expressed based on adequate possibility theory tools;
2. Proposing Bi-ADIPOK as a new Bilevel SBSE method and tool for detecting and identifying code smells under the uncertainty of class labels;
3. Showing our Bi-ADIPOK tool performance upon a group of detailed and statistically analyzed comparative experiments on six commonly open-

source software systems with regard to four relevant approaches as well as the baseline PK-NN [28].

Structure of the paper. Section 2 provides the fundamental concepts regarding the possibility theory. Section 3 presents the basic concepts of the Bilevel Optimization problem. Section 4 defines the principal motivations behind our study and describes our approach Bi-ADIPOK. Section 5 replicate the experimental results of our work in the aim to assess the performance of our approach as well as to compare it with respect to the state-of-the-art approaches. Section 6 details the potential threats that could hinder our experimentations validity. Section 7 discusses the literature related to the code smell detection tools. Section 8 concludes this paper and outlines avenues for future research.

2. Fundamental concepts of possibility theory

Possibility theory is one of the well-known uncertainty theories that proposes a suitable model to represent the imperfect (incomplete and uncertain) information. This theory was developed by Zadeh [30] and improved later by several researchers [31]. In this section, we will present the main concepts needed to understand the proposed approach. For more details about the possibility theory, please refer to [31].

2.1. Possibility distributions

One basic concept in possibility theory is that of a possibility distribution that is denoted by π. Let, Ω be the universe of discourse including various states of the world (i.e., $\Omega = \{\omega_1, ..., \omega_n\}$) and π is a function that associates to each state in Ω (i.e., each ω_i) a value from the unit interval (i.e., [0, 1]) that refers to the possibilistic scale L. The associated value is called a possibility degree and it represents our knowledge regarding a given state of the real world. By convention, $\pi(\omega_k) = 1$ indicates that the achievement of ω_k is completely possible, while ω_k is considered as an excluded state iff $\pi(\omega_k) = 0$. Generally, ω_k is considered somewhat plausible (i.e., flexible) iff $\pi(\omega_k) \in]0, 1[$. Moreover,

ω_k is considered as more plausible (or more specific) than ω_j only for the case where $\pi(\omega_k) > \pi(\omega_k)$. The possibility distribution is considered as a normalized one in case where there exists at least one state from Ω that is fully possible (i.e., $max\ \omega \in \Omega\ \{\pi(\omega)\} = 1$). Otherwise, π (the possibility distribution) is a sub-normalized possibility distribution and this could be demonstrated using the inconsistency degree Inc as follows:

$$Inc(\pi) = 1 - max_{\omega \in \Omega}\{\pi(\omega)\} \tag{1}$$

It is obvious that, in case of normalized π, $Inc(\pi)$ will be equal to 0. The concept of inconsistency is employed to calculate the amount of conflict between two possibilistic chunks of information such as π_1 and π_2. More formally, the amount of conflict could be assessed $Inc(\pi_1 \wedge \pi_1)$ where the conjunction operator (i.e., $\wedge$) is the minimum (min) operator:

$$Inc(\pi_1 \wedge \pi_2) = 1 - max_{\omega_i \in \Omega}\{min_{\omega_i \in \Omega}\{\pi_1(\omega_i), \pi_2(\omega_i)\}\} \tag{2}$$

In possibility theory, two extreme knowledge forms of possibility distributions could be differentiated as follows:

- *Complete knowledge:* $\exists\ \omega_k \in \Omega$, $\pi(\omega_k) = 1$ and all remaining states (ω_i) in Ω (i.e., $\omega_i \neq \omega_k$) s.t. $\pi(\omega_i) = 0$. In this case, one fully possible element exists, while the remaining ones are impossible.

- *Total ignorance:* $\forall\ \omega_k \in \Omega$, $\pi(\omega_k) = 1$. This case indicates that all the states in Ω are possible.

2.2. Information Fusion modes in possibility theory

Conjunctive Fusion: This mode is suitable for the case where all the detectors (information sources) agree with each other. The conjunctive fusion was proposed by Dubois and Prade [32] and it is expressed as follows:

$$\pi_\wedge(\omega) = \otimes_{i=1..n}\pi_i(\omega), \forall \omega \in \Omega \tag{3}$$

were $\otimes$ represents a [0,1]-valued operation specified on [0,1]$\times$[0,1] and $\pi_\wedge$ refers to the conjunctive fusion mode of π. Such fusion mode employs the intersection

with the aim to infer the resulting conclusion. Notably, there exist different operators like *product*, *minimum*, and *Lukasievicz t-norm*:

- Product: $\pi_1 \otimes \pi_2 = \pi_1 \times \pi_2$
- Minimum: $\pi_1 \otimes \pi_2 = min(\pi_1, \pi_2)$
- Lukasievicz t-norm: $\pi_1 \otimes \pi_2 = max(0, \pi_1 + \pi_2 - 1)$

Disjunctive Fusion: This mode of combination is adopted when detectors (information sources) are in disagreement (or conflict). This fusion rule was suggested by Dubois and Prade [32] and it is defined as:

$$\pi_{\vee}(\omega) = \oplus_{j=1..n} \pi_j(\omega), \forall \omega \in \Omega \tag{4}$$

where $\oplus$ represents a [0, 1]-valued operation specified on [0, 1]$\times$[0, 1], and $\pi_{\vee}$ corresponds to the disjunctive fusion mode of π. This mode employs different operators:

- Max: $\pi_1 \oplus \pi_2 = max(\pi_1, \pi_2)$
- Probabilistic: $\pi_1 \oplus \pi_2 = \pi_1 + \pi_2 - \pi_1 \times \pi_2$
- Lukasievicz t-conorm: $\pi_1 \oplus \pi_2 = min(1, \pi_1 + \pi_2)$

2.3. Possibilistic similarity measures

To measure the similarity between different uncertain information sources, the researchers have suggested various similarity measures that are inspired from many theories. For the case of possibility theory, some measures have been suggested like Minkowski distance [33], information closeness [34], Sanguesa et al distance [35], and information divergence [36]. However, all these measures are unable to satisfy the inconsistency criterion. These measures are the main criteria that have been used to measure the possibilistic similarity apart from the distance criterion. To take both criteria (inconsistency and distance) into account, Jenhani et al. [37] defined a new measure called information affinity (denoted by Aff) and it is expressed as follows:

$$Aff(\pi_1, \pi_2) = 1 - d(\pi_1, \pi_2) + Inc(\pi_1, \pi_2) \tag{5}$$

where $Inc(\pi_1, \pi_2)$ refers to the inconsistency (cf. Equation 2) and $d(\pi_1, \pi_2)$ is the distance measure that relies on the normalized Manhatten distance where the formula conducting to this distance computation is defined as follows:

$$d(\pi_1, \pi_2) = \frac{\sum_{i=1}^{n} |\pi_1(\omega_i) - \pi_2(\omega_i)|}{n} \tag{6}$$

To better understand how the Information Affinity measure is calculated, we consider two possibility distributions $\pi_1(0.4,\ 1)$ and $\pi_2(1,\ 0.23)$, which represent two experts opinions in the detection case. As a result, the Affinity is calculated in the following manner:

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

2.4. Discounting

Generally, the information coming from an expert should not be blindly taken into account due to a problem regarding the information reliability. More precisely, the information may not be fully reliable since the human expert may suffer from the lack of expertise regarding a domain, hence, bad measure values could be generated. To take into account the reliability source, a discounting method should be adopted to update the experts' information. Let α be the reliability degree for a given source of information represented by a possibility distribution π and π' be the updated possibility distribution [31] (cf. Equation 7). We notice that if α is fully reliable (i.e., equals to 1), then the source could be fully confident. Based on this fact, π' is not modified ($\pi' = \pi$). Otherwise, π' is modified since the source is not reliable ($\alpha = 0$) and the obtained information will be taken into account as fully imprecise.

$$\pi' = \alpha \times \pi + 1 - \alpha \tag{7}$$

3. Fundamental concepts of Bilevel Optimization

BiLevel Optimization Problem (BLOP) has two optimization levels: (1) the upper level and (2) the lower level [38, 39]. Each level has its own set of constraints, decision variables, and objective functions. As a result, there are two types of variables: upper level variables x_u and lower level variables x_l. It's worth noting that the optimization task for the lower-level problem is carried out with respect to the variables xl, with the variables x_u acting as parameters. As a result, each x_u corresponds to a different lower level problem for which an optimal solution must be found. The upper level problem considers all variables (x_u and x_l), and optimization is expected to be conducted with both sets of variables (cf. Fig. 2). The following definition gives the analytical formulation of a BLOP:

$$
\begin{aligned}
& Min_{x_u \in X_U, x_l \in X_L} F(x_u, x_l) \\
\text{s.t.} \quad & \begin{cases} x_l \in ArgMin \left\{ f(x_u, x_l), \right\} g(x_u, x_l) \leq 0, \\ G(x_u, x_l) \leq 0, i = 1...l, j = 1...J \end{cases}
\end{aligned} \tag{8}
$$

where g_i denotes the lower-level constraint set and G_j denotes the upper-level constraint set. Due to the problem's structure, only the best solutions from the lower level optimization task may be considered as viable candidates for the upper level optimization task. For example, at the upper level, a member $x^1 = (x_u^1, x_l^1)$ is only feasible if x^1 satisfies the upper level constraints and the lower level problem referring to x_u^1 has an optimal solution, x_l^1. Therefore, the upper-level problem makes the decision first. The lower level must then decide, having known the upper level's decision, in order to optimize its own objective function without regard for the upper objective function. The x_u and x_l are decision variables that can be continuous, discrete, or mixed.

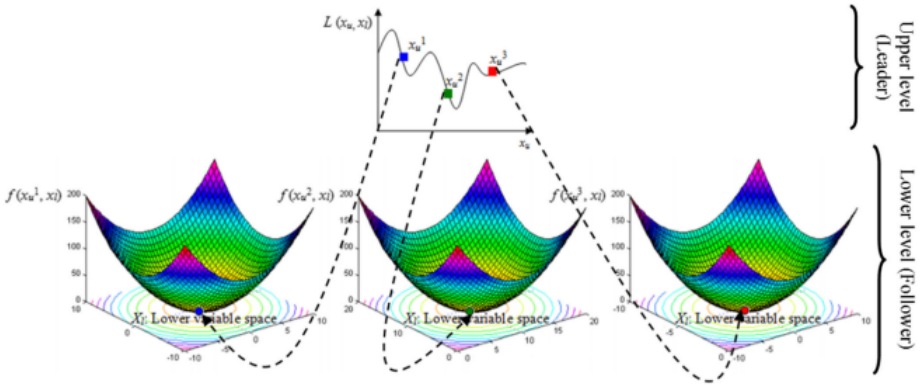

Figure 2: An illustration of the two levels of a bilevel single-objective optimization problem [40].

4. Bi-ADIPOK: Bilevel Anti-pattern Detection and Identification using Possibilistic Optimized K-NNs

In this section, we present the principal components of our proposed approach used for code smell detection and identification under uncertainty, called Bi-ADIPOK. The uncertainty is mainly inherent from the subjectivity and the uncertainty of experts perceptions regarding the smelliness of a software project class as well as the existing smell types. In fact, ignoring the uncertainty may result in a decrease in the quality of the results produced by detectors. We noticed that detection and identification tasks are done using the same mechanism, but using different BEs characterized by uncertain class labels. More precisely, the uncertainty of the experts' opinions towards class labels is represented in the form of likelihood values called possibility degrees. These latter are built based on five probabilistic classifiers (Naïve Bayes classifier [41], Probabilistic K-NN [42], Bayesian Networks [43, 44], Naïve Bayes Nearest Neighbor [45], and Probabilistic Decision Tree [46]) in the aim to attribute the aggregated probability distributions for every existing instance (software class) in the Base of Examples (BE). The adoption of probabilistic classifiers is an extension of the idea proposed in [47] where a set of classifiers are employed to mimic the experts. In

other words, in this study, the chosen classifiers are used in the aim to mimic the software engineers' uncertainty for labeling the software classes within the BE. This will aid the software engineer to model its uncertainty regarding the smelliness of a software class in the form of likelihood values.

Hereafter, these produced values are converted into possibility distributions via conversion formula (described in detail below). For the detection task, the BE may incorporate smell types or vice versa, while for the identification task, the BE contains only one smell type. Therefore, the identification task is seen as a subcase of the detection task, where the approach only operates on a single smell type. To promote the understanding of the working principle of Bi-ADIPOK, we first introduce the main needs behind its requirements. Second, we describe the artificial building of the PBEs. Third, we clearly present the global schema of our detection method within an uncertain environment. Fourth, we clearly state how the GA is used for the evolution of detectors (i.e., PK-NNs). We mention that PK-NNs are used in our work because they are suitable for learning from instances (software classes) with uncertain class labels. In the remainder of this section, we tend to describe the solution encoding, the fitness function, and the reproduction (i.e., crossover and mutation) operators. Finally, we tend to illustrate how the resulted detectors could be used for both detection and identification problems.

4.1. Artificial creation of Possibilistic BEs

Similar to many data mining fields, the SE (Software Engineering) industry could be affected by the uncertainty issue. Indeed, the BE may be overwhelmed by uncertain class labels. The principal sources of uncertainty may be related to: (1) the lack of knowledge of human experts and/or (2) the subjectivity of their conflicting opinions. In this uncertain environment, experts can express their opinions in the form of possibility distribution where each possibility degree refers to the degree of membership of every software class to every class label. To be able to produce possibilistic code smell detectors, we need a group of PBEs. In these PBEs, every instance (i.e., software class) could be attributed

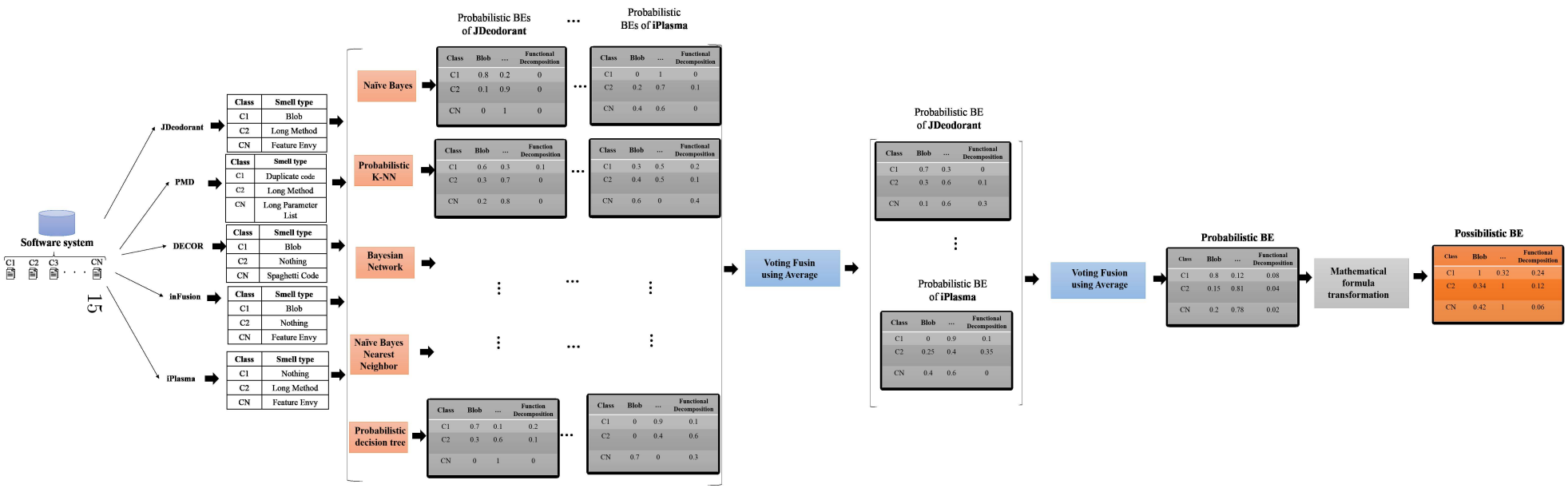

Figure 3: Possibilistic BE generation process.

to a group of class labels and every class label has a possibility degree. Nevertheless, the existing BEs do not include any uncertain class labels, which could be used as training datasets for the induction of possibilistic detectors. Since the aim of this work is to only deal with uncertainty at the level of class labels and not at the level of attributes, only the class labels are replaced with possibility distributions; while the original features' values are kept. The possibility distributions are produced by an opinion-based classifier (extended from [47]) as well as a conversion formula [48] (cf. Figure 3). The conversion process can be described as follows:

1. First, we adopted five different advisors (DECOR [49], JDeodorant [50], inFusion[1], iPlasma[2], and PMD [51]) to produce five crisp BEs.
2. Second, we run autonomously five probabilistic classifiers (i.e., Naïve Bayes classifier, Probabilistic K-NN, Bayesian Networks, Naïve Bayes Nearest Neighbor, and Probabilistic Decision Tree). Then, we use the average operator twice to apply the voting fusion. For the first time, we use the voting fusion in the aim to aggregate the resulted probabilistic BE for every classifier. For the second time, it was used to get a single probabilistic BE (cf. Figure 3).
3. Finally, we employed the following conversion formula proposed by [48] to convert the obtained probability distributions into possibility ones:

$$\pi_i(\omega_i) = i \times p(\omega_i) + \sum_{j=i+1}^{n} p(\omega_j), \forall i = 1...n \tag{9}$$

where the probability distribution p defined on Ω should be sorted in descending order ($p(\omega_1) \geq p(\omega_2) \geq ... \geq p(\omega_n)$) before starting the transformation of p into π. It should be noted that the sum of the probability distribution degrees should be equal to one.

[1] http://www.intooitus.com/products/infusion
[2] http://loose.upt.ro/iplasma/

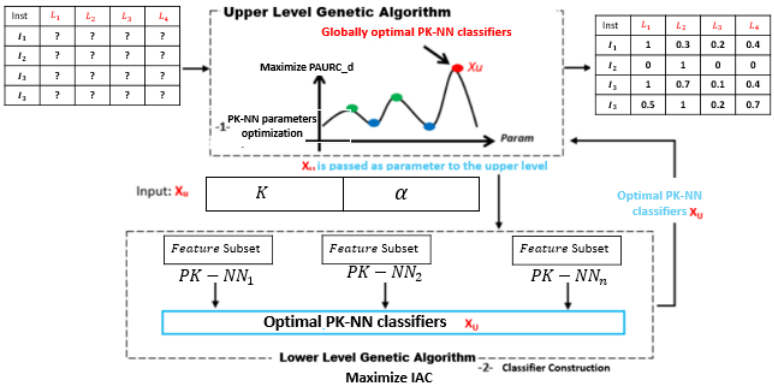

Figure 4: The global schema of the Bi-ADIPOK approach.

4.2. Main schema

Figure 4 shows the main architecture of the Bi-ADIPOK method, which is predominantly made out of two parts: (1) the possibilistic smell detectors generation module and (2) the possibilistic smell detectors application one. The former is divided into two levels that are:(1) the upper level and (2) the lower level. The former generates a set of optimized PK-NNs parameters, while the lower level optimizes a set of PK-NNs based on the parameters obtained from the upper level. The gnerated detectors are assembled into a single base of examples. However, through the identification process, Bi-ADIPOK yields a group of specialized detectors (i.e., a set of specialized PK-NNs) that are trained on the chosen smell type. As a result, we get a base of detectors for every considered smell type (i.e., Data class, Feature Envy, Blob, etc.). Therefore, the practitioners can analyze any unseen software's classes, more precisely, these latter are checked based on the obtained set of detectors for each smell type where their outputs are aggregated based on a specific majority voting strategy to determine the existing smell types. More details about this process are presented later.

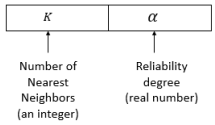

Figure 5: Solution encoding employed in the upper level.

4.2.1. Upper level optimization

The optimization process is carried out as follows at this level. The proposed approach starts with the solution with the lowest fitness value and gradually increase the fitness values of the remaining solutions until we reach the stopping criterion. The mating selection and variation operators (crossover and mutation) are then applied. We have used PAURPC_d as a fitness function to assess the solutions. Such measure is able to deal with the problem of uncertain and imbalanced data issues. Furthermore, the PAURPC_d measure promotes upper-level convergence and diversity, which aids the algorithm in approximating optimal solutions.

- Solution Encoding: At the upper level, a candidate parameters values solution is represented as a chromosome in which every case corresponds to a parameter value.It is important to know that each individual is expressed by a vector containing 2 numbers where the former is an integer value that corresponds to the parameter K and the second value is a float (varies between 0 and 1) that corresponds to the parameter α. Figure 5 shows an example of a solution of PK-NN parameters optimization.

- Solution evaluation: The solution evaluation step evaluates the classifiers' performance,guiding the evolution to individuals who have high performance because they considers the uncertain and imbalance problems at the same time. In fact, before the evaluation of each individual (PK-NN), the K nearest neighbors of the current individual are selected using the Euclidean distance. Then, a combination of the possibility degrees for every class label should be performed to produce a final possibility distribution for the new unseen instance using Equation 10. Then, the obtained

information (possibility distribution) for the unseen software class is updated based on α (i.e., the reliability of the detector that exists in the vector presentation) using Equation 7.

$$\forall \omega_q \in \Omega, \pi_{\sum}(\omega_q) = \frac{\sum_{i=1}^{K} \pi(\omega_q^{(i)})}{\sum_{i=1}^{K} \pi(\omega^{(i)})} \tag{10}$$

where ω_q represents the class of the ith instance among the K chosen nearest neighbors. Suppose that we have three nearest neighbors from the BE of a given unseen instance with the following possibility distributions ([1 0.5], [0.3 1], [1 0.5]). Using Equation 10, the possibilistic distribution of the unseen software class will be equal to [1 0.87], which is computed as follows: The first bound is computed as the sum of 1, 0.3, and 1, while the second bound is computed as the sum of 0.5, 1, and 0.5. Then, we will normalize the two obtained bounds by dividing them on the max bound value (i.e., 0.534). A variety of fitness functions have been proposed such as the accuracy measure [52]. Notwithstanding, the use of the existing measures as a fitness function within the case of uncertain class labels is a crucial problem. As we mentioned earlier, the PK-NN produces possibility distributions through the process of classification. However, these metrics give a great deal of attention to the most plausible class labels (i.e., their possibility degrees equal to 1), whereas the rest are overlooked. In fact, ignoring some possibility degrees (i.e., class labels) may result in the loss of a large amount of produced information by detectors. In addition to the uncertain class labels issue, the code smell detection problem is considered to be an imbalanced data classification problem [53, 4], where BE is made out of two sub-sets: (1) the majority class and (2) the minority one. The latter cardinality is much less than the former one, which involves an imbalance problem that should be considered by the detector. To address the problem of skewed (imbalanced) data, our developed fitness function is based on $mAURPC - OVA$ (modified Area Under Recall Precision Curve-One Versus All) [54]. Our choice could be justified by the fact that the $mAURPC - OVA$ is insensitive to the data imbalance problem

especially in the case where the number of Non-smelly (negative) examples largely surpasses the number of smelly (positive) examples. More precisely, the $mAURPC - OVA$ metric depicts how well a detector can classify smelly examples that are truly smelly examples. However, this measure is unlikely to be suitable for the uncertain environment and will diminish the overall performance of the detector. Motivated by this observation, in this paper, we have proposed a new performance measure called $PomAURPC - OVA_dist$ (Possibilistic modified Area Under Recall Precision Curve- One Versus All_distance) that is defined by equation 17. The computation of $PomAURPC - OVA_dist$ relies on the mean distances between the predicted (π^{pred}) and the actual possibility distribution (π^{act}) of every unclassified (unlabeled) software class $\overrightarrow{I_j}$. In fact, when $PomAURPC - OVA_dist$ is near 1, the obtained detector is precise and the produced possibility distributions have high-quality and faithfulness compared to real (actual) ones. On the contrary, if an assigned fitness function (relaying on $PomAURPC - OVA_dist$) to a detector drops to 0, then the detector is considered terrible. We notice that due to the reason of clarity and space, we prefer to denote $PomAURPC - OVA_dist$ as $PAURPC_d$. The $PAURPC_d$ could be expressed as follows: The $Sd(\overrightarrow{I_j})$ (Similarity distance) represents the distance between the generated possibility distribution (π_{pred}) and the actual possibility distribution (π_{act}). This measure belongs to [0, 2] and it is calculated as follows:

$$Sd(\overrightarrow{I_j}) = \sum_{i=1}^{|C|} (\pi^{pred}(C_i) - \pi^j(C_i)) \tag{11}$$

The $SD(\overrightarrow{I_j})$ measure, has values between 0 and 1, is a modification performed on the $Sd(\overrightarrow{I_j})$ in the aim to have a significance close to $mAURPC - OVA$ and it is computed as follows:

$$SD(\overrightarrow{I_j}) = 1 - \frac{Sd(\overrightarrow{I_j})}{2} \tag{12}$$

The $TP_{_dist}$ is the sum of the distances of the Actual Smelly classes cor-

rectly classified and it is calculated as follows:

$$TP_{_dist} = \sum_{\overrightarrow{I_j} \in \text{ASCcc}} SD(\overrightarrow{I_j}) \tag{13}$$

where $ASCcc$ is the Actual Smelly classes correctly classified. The $TN_{_dist}$ is the sum of the distances of the Actual Smelly classes correctly classified and it is calculated as follows:

$$TN_{_dist} = \sum_{\overrightarrow{I_j} \in \text{ANSCcc}} SD(\overrightarrow{I_j}) \tag{14}$$

where $ANSCcc$ is the Actual Non-smelly classes correctly classified. The $FP_{_dist}$ is the sum of the distances of the Actual Smelly classes miss-classified as Non-smelly and it is calculated as follows:

$$FP_{_dist} = \sum_{\overrightarrow{I_j} \in \text{ASCmNs}} SD(\overrightarrow{I_j}) \tag{15}$$

where $ASCmNs$ is the Actual Smelly classes miss-classified as Non-smelly. $FN_{_dist}$ is the sum of the distances of the ctual Non-smelly classes miss-classified as Smelly and it is computed follows:

$$FN_{_dist} = \sum_{\overrightarrow{I_j} \in \text{ANSCms}} SD(\overrightarrow{I_j}) \tag{16}$$

where $ANSCms$ is the Actual Non-smelly classes miss-classified as Smelly. The $PAURPC_d$ is calculated based on the $TP_{_dist}$, $FP_{_dist}$, $TN_{_dist}$, and $FN_{_dist}$ as follows:

$$\begin{aligned} PAURPC_d = \frac{1}{2 \times |C|} \times ((&\frac{\frac{TP_{_dist}}{(TP_{_dist}+FN_{_dist})}}{\frac{TP_{_dist}}{(TP_{_dist}+FN_{_dist})} + \frac{FP_{_dist}}{(FP_{_dist}+TN_{_dist})}} + \\ &\frac{TP_{_dist}}{(TP_{_dist} + FN_{_dist})}) + (\frac{\frac{TN_{_dist}}{(FP_{_dist}+TN_{_dist})}}{\frac{FN_{_dist}}{(TP_{_dist}+FN_{_dist})} + \frac{TN_{_dist}}{(FP_{_dist}+TN_{_dist})}} + \\ &\frac{TN_{_dist}}{(FP_{_dist} + TN_{_dist})})) \end{aligned} \tag{17}$$

Based on the obtained position of the most plausible classes, the $(\overrightarrow{I_j})$ is added to the adequate quantity (i.e., $TP_{_dist}$or $FP_{_dist}$ or $TN_{_dist}$

or $FN_{_dist}$). Thsese distances are computed to obtain at the end the $PAURPC_d$. For example, if π_{act}=[1 0.4] and π_{pred}=[1 0.2], then based on the calculated distance between those possibility distributions, we will add the obtained distance to the $TP_{_dist}$. Conversely, if π_{pred}=[0.2 1], then the obtained distance will be added to $FP_{_dist}$.

- Mating selection operator: As noted earlier, one of the strongest points of our Bi-ADIPOK method consists of its capability to avoid the local optima and to reach the globally optimal smell detectors. The principal mechanism that guarantees such behavior is the mating selection operator that we have adopted, i.e., the binary tournament selection operator [55], which can be defined as follows:
  - First, we select $(N/2)$ parents for the phase of reproduction, N represents the population size.
  - Then, we execute a loop of $(N/2)$ iterations. More clearly, two individuals (PK-NNs) are selected arbitrarily.
  - Finally, the fit parent is retained and added to the mating pool.

  Such selection strategy makes possible the selection of good and bad individuals with preferential towards good individuals.

- Crossover and mutation operators: As shown by Figure 4, we employed the SBX crossover operator with rounding up mechanism [56] since the SBX outputs real numbers and the K value should be an integer, the obtained value from the SBX is rounded up to the nearest integer (cf. Figure 6). For the mutation operation, we performed the mutation similar to the crossover operation (i.e., per part). As illustrated by Figure 5, we adopted, for the first part, the one-point mutation operator [57], while for the second part we used the polynomial mutation [56] with rounding up as the value of the K parameter is an integer one (cf. Figure 7). However, we applied, for the last part, a polynomial mutation without rounding up due to the real type of the parameter (cf. Figure 7). It is important to

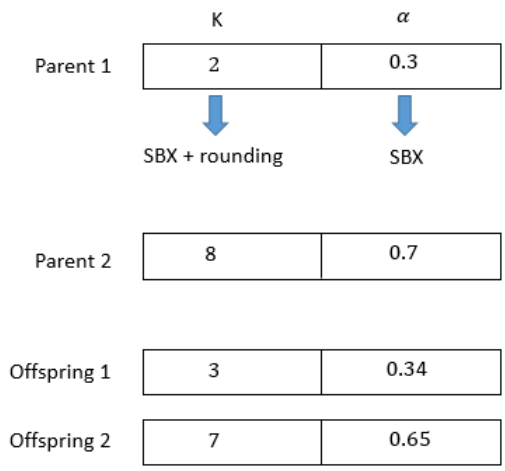

Figure 6: Crossover operator used in the upper level.

note that once the mutation is triggered, it modifies the three chromosome parts simultaneously.

4.2.2. Lower level optimization

At this level, an upper level solution is passed as a fixed parameter to the lower level. Then, a set of PK-NN solutions are generated in the lower level. After that the fitness values of the lower level PK-NN individuals are computed (i.e., the IAC fitness values are computed). The fittest solutions in the lower level will be assigned to the upper level solution. Finally,the set of PK-NN individuals are evolved based on the variation operators.

- Solution Encoding: At the lower level, each chromosome (cf. Figure 8) encodes the feature set (FS) parameter, which is a binary vector of structural metrics (cf. Appendix 10). In fact, the weights of the selected metrics are assigned the '1' values, while the weights of the remaining metrics took '0' values. The PK-NN will be formed by merging the solution of the lower level with the generated parameter of the upper level.

- Solution evaluation: The evalaution of the PK-NNs at the lower level has been performed using the IAC (Information Affinity-based criterion)

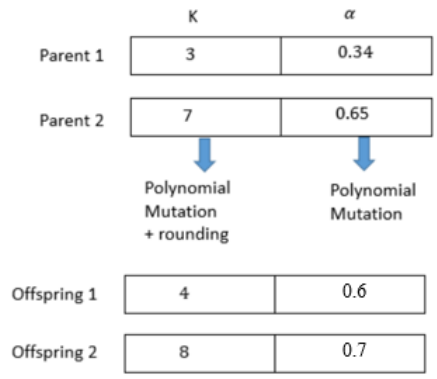

Figure 7: Mutation operator adopted in the upper level.

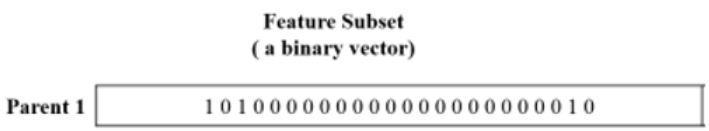

Figure 8: Solution Encoding used in the lower level

(cf. Equation 18) which was suggested by [25]. Such a measure relies on the Affinity distance (cf. Equation 5) to compute the distance between two possibility distributions: the original (π^{act}) and the predicted one (π^{pred}). In fact, the IAC has been considered since it considers the uncertainty aspect. In fact, when the values of the are close to 1 means that the obtained detectors are more accurate than the original ones, while the provided possibility distributions have high qualities and faithfulness. Nonetheless, when the values of the IAC fall to 0, this implies that the evolved detectors are weak.

$$IAC = \frac{1}{n}\sum_{i=1}^{n} Aff(\pi_i^{act}, \pi_i^{pred)} \tag{18}$$

- Mating selection operator: At hte lower level, the tournament selection operator is used to select parents for the reproduction. It is important to know that at every generation, half of the population (i.e., N/2) is

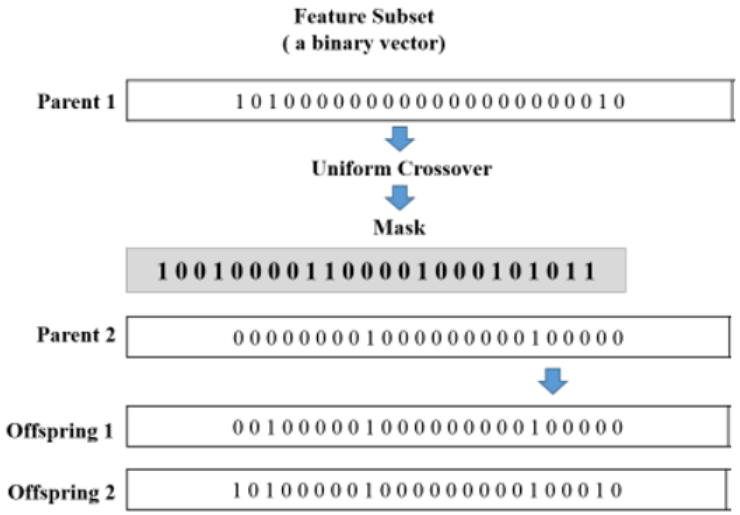

Figure 9: Crossover operator adopted in the lower level

selected.

- Crossover and mutation operators: For the variation of the population at the lower level, we adopt the uniform crossover [58] for the first part (binary string), which in turn employs a random binary mask (cf. Figure 9). For the mutation operation, we adopted, the one-point mutation operator [57] (cf. Figure 10).

4.3. Optimized possibilistic detectors use

Once the optimized possibilistic smell detectors (PK-NNs) are generated using the GA, the Bi-ADIPOK method is ready to be employed on unseen software classes in the aim to detect the occurrences of code smells and/or identify their types. More clearly, the software engineers are permitted to use some, or all, of the best detectors. It is recommended to employ all the produced smell detectors following the voting fusion process as demonstrated by Figure 11. Indeed, every smell detector (PK-NN) corresponds to possibility distribution (obtained from the K nearest neighbors), where each possibility degree refers to the degree

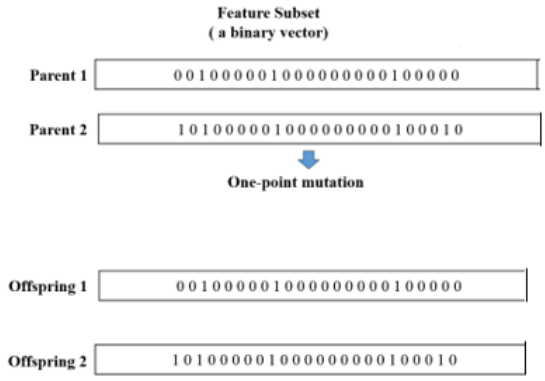

Figure 10: Mutation operator adopted in the lower level

of membership to each BE' class label. Therefore, the AFO (Adaptive Fusion Operator) [29] (cf. Equation 19) is adopted to aggregate the different generated decisions (i.e., possibility distributions) from the set of smell detectors. For more comprehension, the AFO combines the possibility distributions (i.e., decisions) based on the disjunctive fusion operator (cf. sub-section 2.2), especially where the detectors are in disagreement. However, the conjunctive fusion operator (cf. sub-section 2.2) is adopted by the AFO for combining the decisions where the sources (smell detectors are in agreement). The agreement and disagreement state is determined based on the amount of conflict between the produced decisions (possibility distributions) by the detectors. Thus, if $Inc(\pi_1 \wedge \pi_2) = 0$ this indicates that the code smells are in the state of agreement, while the state of the detectors could be in disagreement when $Inc(\pi_1 \wedge \pi_2) \neq 0$. Figure 11 outlines the detection of smelly software classes according to PBE, which also contains both smelly and non-smelly examples. For more accuracy about the smell types, Figure 11 illustrates the identification process where the Bi-ADIPOK tool places a specific base of smell detectors for every considered smell type.

$$\forall \omega \in \Omega, \quad \pi_{\ AD}(\omega) = max(\pi_{\wedge}(\omega),\ min(\pi_{\vee}(\omega),\ 1 - h(\pi_1, \pi_2))) \qquad (19)$$

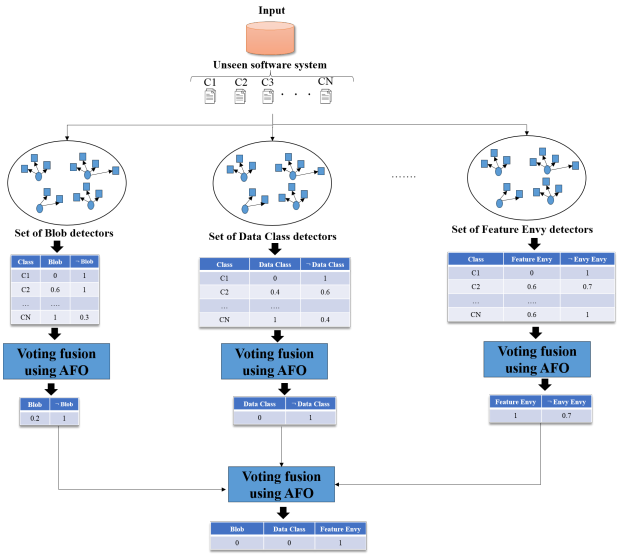

Figure 11: Illustrating the use of Bi-ADIPOK for smell type identification on an unseen software class.

$$Inc(\pi_1 \wedge \pi_2)) = 1 - max(min(\pi_1, \pi_2)) \tag{20}$$

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 $h(\pi_1(\omega), \pi_2(\omega))$ depicts the agreement degree among two smell detectors [59]. Note that $\pi_\wedge$ and $\pi_\vee$ refer to the conjunctive and disjunctive operators, respectively.

5. Experimental validation

The overall performance of Bi-ADIPOK is assessed based on the most outstanding existing works in the literature. Over a series of comparative experiments employing six commonly used software projects:

- **RQ1:** How does our Bi-ADIPOK work in an uncertain setting for the code smell detection issue? To answer this question, we report the performance of using a suitable fitness function to simultaneously handle uncertain class labels and data imbalance issues. In addition, we assess the effectiveness

of Bi-ADIPOK for the adaptation of appropriate detectors (PK-NNs) with a view to the processing of possibilistic class labels. These are performed comparing to four state-of-the-art approaches through a serious of comparative experiments.

- **RQ2:** How does our Bi-ADIPOK process identify the existing code smell types? We report the performance of employing a set of optimized detectors for every smell type to identify the existing smell types in order to answer the current research question. Therefore, our Bi-ADIPOK is compared to four respective state-of-the-art approaches.

- **RQ3:** How does our Bi-ADIPOK approach detect/or identify code smells for uncertain class labels and data imbalance issues compared to the baseline PK-NN approach? It is important to know how far our proposed approach could go beyond the baseline PK-NN with a greedy search.

5.1. Subject Systems

Our Bi-ADIPOK tool is assessed on the basis of a series of commonly adopted open-source Java projects, consisting of Xerces-J[3], GanttProject[4], ArgoUML[5], Ant-Apache[6], JFreechart[7], and Azureus[8]. Table 1 lists the features of the software projects under consideration in our experimental analysis, where the columns (from the left to the right) display the project name, the release number, the description, the size in relation to the number of classes (NOC) as well as the number of lines of code (KLOC: thousands of code Lines), respectively.

JFreechat is a professional Java chart library dedicated to the production of high-quality charts. GanttProjects is a cross-platform dedicated to

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

Table 1: List of considered Software Projects

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

scheduling projects. The ArgoUML tool is an open-source platform for modeling UML. Ant-Apache is an open-source Java library and a built-in tool conceived for the Java applications. Xerces-J is an open-source project devoted to the parsing of XML files. Azureus is a P2P (i.e., end-to-end) platform for sharing files inter users. These systems have been considered for some reason. First, they are open source and their source code is publicly accessible. The choice of such open-source software projects enables this work to be re-evaluated by other researchers. Second, all of them are wealthy in terms of the number of existing code smells as well as they are frequently used within the empirical smell detection works [12, 60, 8]. Third, various developer teams have designed the projects under consideration and can, therefore, decrease the bias of specific developers. In this experimental study, the constructed PBE is regarded as our ground truth because it is an aggregation of different simulated subjective and uncertain opinions of different experts (probabilistic classifiers). These opinions are quantified in order to take the form of probability values.

5.2. Used baseline approaches

For the comparison of Bi-ADIPOK against the state-of-the-art approaches, four relevant baselines have been selected, these are GP [61], MOGP [62], BLOP [40], and DECOR [49]. The choice of these approaches to baseline was based on two main reasons. From one point of view, the search-based methods are divided into three categories: (1) Mono-objective techniques (GP), (2) Multi-objective techniques (MOGP), and (3) the Bi-level ones (BLOP). Additionally,

these methods permit us to have a more extensive overview of how our approach works against the existing approaches. From another point of view, DECOR is selected as the representative of the heuristic rule-based category; we included it as a baseline because of our willingness to comprehend whether Bi-ADIPOK can actually perform a basic detection approach according to heuristics that are principally calculated on the basis of source code metrics values. A concise synthesis of every baseline approaches' functioning principle (listed above) is Indicated in the following:

- GP: This method uses the genetic programming metaheuristic, which communicates with a BE of code smells to evolve a set of IF-THEN rules. Every solution is referred to as a detection rules' tree; where the internal nodes encompass the structural metrics as well as their corresponding thresholds, whereas the class labels are indicated by the leaf nodes. Compared with the expected number of defects in the BE, these rules have been evolved by the maximization of the number of detected defects. The authors highlighted that GP has achieved an average F-measure of 88% on six software projects, while only three smell types were considered. In the same paper, a multi-objective refactoring approach relying on NSGA II has been proposed to eliminate the detected smells as much as possible.

- MOGP: This method is chosen as a representative of the multi-objective techniques. The MOGP adopts the same encoding as GP, whereas the difference appears in the used fitness function. Additionally, the MOGP has a similar framework to GP and evolves trees by optimizing two conflictual objectives. The first objective is to maximize the detection of the inherent code smells in the BE, whereas the other objective is to reduce the detection pieces of well-designed code fragments. To achieve this aim, NSGA-II interacts with two BEs; the former has smells, while the latter has well-designed code fragments. The authors explained the employment of well-designed codes on the grounds that the use of smells would not

allow all smells to be covered. By maximizing the distance from a well-designed code example, a piece of code can be considered as a suspicious anti-pattern. Based on the reported precision and recall results, the mean F-measure for seven software systems is around 87% when taking into account five types of smells.

- BLOP: This method was suggested in the aim to tackle the lack of diversity issue that could have a BE. For this purpose, the code smell detection problem has been proposed to be modeled as a bi-level optimization one as follows. A set of smell detection rules has been evolved at the upper level, while a set of artificial code smells has been evolved at the lower level. From the fitness viewpoint, the detection of true code smells and artificially generated ones are maximized by the upper level, whereas the probability that the upper-level rules would detect code smells is minimized by the lower level. Therefore, the competition among the two levels is aimed at: (1) generating rules with significant detection capability, and (2) producing unknown artificial code smells to achieve a BE diversification. Based on the reported precision as well as recall values, the average F-measure value of the nine software projects is approximately 90% relative to the seven considered smell types.

- DECOR: This method is not a search-based solution but a heuristic approach. In particular, DECOR employs a collection of rules, known as the "rule card", to describe the inner properties of a sell-affected class. For example, a software class is defined as Blob if it includes LCOM5 (Lack of Cohesion Of Methods) [63] is greater than 20, a number of methods and attributes greater than 20, a suffix in the list { *"Process"*, *"Command"*, *"Control"*, *"Manage"*, *"System"*, *"Drive"*} and one-to-many association with various data classes. The DECOR's ability to identify code smells has been demonstrated with a mean F-measure around of 80% [49].

Table 2: The default parameters configuration.

Parameters	Bi-ADIPOK	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

5.3. Parameter adjustment for Bi-ADIPOK

The adjustment of algorithm parameters is an important aspect that is often overlooked in metaheuristic search algorithms. It is essential to know that the setting of parameters can significantly affect an algorithm's performance on a specific problem. Accordingly, the default Bi-ADIPOK parameters used in the simulation part (cf. Table 2) are adjusted by trial-and-error method [64, 65], which is currently standard practice in both evolutionary computation and SBSE domains [66, 67, 68]. For a fair comparison, the same stopping criterion has been used for methods being compared including ours. Based on this fact, every run has been halted after reaching 256,500 fitness evaluations. This choice is appropriate for all compared approaches as well as BLOP in which two populations are used: (1) the upper level and (2) the lower one. In fact, a population of 30 individuals is evolving at both levels, more precisely, 15 and 19 generations are performed at the upper level and lower one, respectively. These values have been set in the aim to approximate the optimal lower level population, which is necessary for computing the fitness of its corresponding population of the upper level. Accordingly, all algorithms as well as BLOP could thus fulfill the stopping criterion, because under these settings, the number of evaluations carried out by BLOP is (30*15*30*19)= 256,500.

5.4. Performance metrics

When addressing an uncertain data classification problem, the adopted performance metrics should be able to measure the performance of the various approaches taken into account in our study. However, the existing approaches ignore the fact of uncertainty, which may have an adverse effect on their performance. In order to address this issue in our work, we have chosen two suitable measurements for the uncertain environment. The first one is $PAURC_d$ and the second one is the IAC, which have already been introduced above. In fact, our experiments are performed in a within-project manner by using the hold-out validation technique where 70% of the BE is for the training and the remaining 30% is devoted for the performance testing

5.5. Adopted statistical testing methodology

Generally, all the approaches that rely on the GAs generate distinct results from one run to another on the same software system (or problem). It is important to note that the GAs are characterized by stochastic behaviors. In this case, it is difficult to compare stochastic code smell detection approaches because the output (result) can vary in each run. To alleviate the statistical nature of outcomes, the use of statistical tests have been recommended by the researcher in the aim to detect the difference among the provided results [69, 70]. There exist two types of tests: (1) Parametric tests that only work with normalized data and (2) Non-parametric ones. To avoid the data normality problem in this work, the Wilcoxon test [71] have been chosen to be used by carrying out a pairwise comparison. Accordingly, two hypotheses are considered: (1) H_0 indicates that the two median values belonging to the two compared algorithms have no significant difference in the number of runs, while H_1 indicates the opposite. Over this work, the significance rate has been fixed to 5%, which implies that the probability of refusing H_0 is just 0.05. In addition to the significance, it is also important to quantify the results of the compared algorithms that's why the effect size must be reported. Indeed, the Wilcoxon test only permits for verification of whether the generated results are statically different. Such a test

does not give any indication about the difference magnitude. To deal with this problem, the solution is to adopt the Cohen's d statistic [71] since it is a suitable for measuring the effect size that could be: (1) large if d $\geq$ 0.8, (2) medium if 0.5 $\leq$ d $<$ 0.8, (3) small if 0.2$\leq$ d $<$ 0.5, and (4) very small if d $<$ 0.2.

5.6. Analysis of the results

This subsection is intended to report and explain the gathered comparative results with the intention to address all the presented research questions over this work and illustrate the effects of the key features of the Bi-ADIPOK approach. More precisely, those features consist of: (1) the adoption of possibilistic detectors (i.e., PK-NN) for the detection of code smells, (2) the use of the metaheuristic technique, in particular GA, to flee local optima, and (3) the well-orchestrated smells detectors. Additionally, we illustrated the usability of Bi-ADIPOK for the detection case as well as the identification one.

5.6.1. Results for RQ1

In response to RQ1, we perform a number of analyzes on the six chosen software projects with taking into account the uncertainty aspect. This aspect mainly appears in the subjectivity of the experts' opinions regarding the decision about the smelliness of a software class and /or the smell type. To simulate such uncertainty, we have transformed the original BE, which is a certain one (or crisp one), into a possibilistic BE having possibility distributions over its class labels. For more details about the transformation process, please refer to Section 4.1. Over this work, we aim to demonstrate that our Bi-ADIPOK performs well in both cases. The first case is devoted to present the Uncertainty Level (UL) in which half part of the content (i.e., instances' class labels) in the BE are covered by uncertainty. This case represents the scenario where UL=50%. The second case is dedicated to the crisp BE (i.e., UL=0%), where all the content (i.e., instances' class labels) are certain. Based on Table 3, Bi-ADIPOK surpasses all the chosen state-of-the-art approaches with a PAURC_d raging from 0.902 and 0.932. With a PAURC_d varying between 0.15321 and 0.224, the BLOP

approach is ranked second. In contrast, the remaining search-based approaches (i.e., GP and MOGP) shown weak results, with the highest PAURC_d values being 0.1611 and 0.213, respectively. The outperformance of Bi-ADIPOK upon the rest of approaches (i.e., GP, and MOGP, and BLOP) can be clarified by the fact that our Bi-ADIPOK approach takes into consideration the uncertainty aspect in the solution assessment process, whereas this aspect is often ignored by the competitors approaches. We note that the used fitness function by Bi-ADIPOK (PAURC_d) has shown its effectiveness in addressing the problem of uncertain class labels because it is insensitive to the data imbalance issue. However, DECOR has shown the worst results since its PAURC_d values are between 0.146 and 0.429. The results of DECOR can be proved to be reasonable since the adopted rules are drawn up manually and without taking into account the uncertainty factor. As for the results of the IAC metric, they are almost like the results displayed by the PAURC_d as the certain case is a sub-case of the uncertain one. It should be noted that the certain (crisp) case coincides with the ground of the truth. This means that a crisp case might be depicted by a binary vector containing only a value of 1 that refers to the real class label whereas the non-real class labels' are set to 0. The shown results prove that Bi-ADIPOK is able to detect effectively under uncertain environment. According to Table 3, the obtained statistical results for the detection case reveal that our approach (i.e., Bi-ADIPOK) is more significant than DÉCOR, GP, MOGP, and BLOP due to the consideration of the uncertainty of human experts by means of detector method as well as the adopted fitness function.

Table 4 lists the obtained results of the adopted measures for the five smell detection approaches in the case of a crisp environment, which UL=0%. We recall that the crisp environment corresponds to the case where the BE class labels are certain. Accordingly, the PAURC_d performance behaves similarly to the mAURPC-OVA. This table shows that the PAURC_d rate of our Bi-ADIPOK approach is between 0.928 and 0.957, which is better than the other chosen approaches. The second-best approach is doled out to the BLOP approach since its PAURC_d lies between 0.554 and 0.337. The rest of the approaches (i.e.,

DECOR, GP, and MOGP) shows poor results as the maximum of their displayed PAURC_d values are 0.351, 0.401, and 0.531, respectively. The reason why Bi-ADIPOK surpassed its rivals can be clarified as follows. The fitness functions used by the chosen approaches (without considering ours) are not suitable to handle the data imbalance issue, which can trick the search process for the obtained smell detection rules. On the contrary, the Bi-ADIPOK fitness function is not sensitive to the data imbalance issue, because the behavior of PAURPC_d is similar to mAURPC-OVA, particularly in certain environments. Based on Table 4, the statistical results for the detection case under certain environment reveal that our approach (i.e., Bi-ADIPOK) is more significant than DÉCOR, GP, MOGP, and BLOP due to the fact that our approach considers the problem of imbalanced data through the adopted fitness function.

In summary, the Bi-ADIPOK performance improvement could be clarified as follows. For the case of uncertain setting, the PAUPRC_d is an adequate metric for dealing with the uncertainty inherent in the BE class labels. In contrast, the PAUPRC_d identically behaves to the mAURPC-OVA in the case of certain setting since the mAURPC-OVA is has been proven to be insensitive to the problem of imbalanced data.

5.6.2. Results for RQ2

To answer RQ2, we aim in this part to assess the overall performance of the compared approaches for the identification of the smell type issue for the uncertain environment as well as the certain one. We recall that the uncertain setting refers to the case where a part of the BE class labels is uncertain, while in a certain setting, the whole instances' class labels in the BE are certain. The identification process is considered as a harder process (more than the detection one) since its data imbalance ratio is greater than the imbalance ratio of the detection process. To achieve the comparative experiments, all software projects have been combined into a single BE and then the minority class for every smell type has been determined. However, we have noticed in this task that the number of classes that are not smelly is greater than that of smelly

Table 3: PAURPC_d and IAC median scores of Bi-ADIPOK, DECOR, GP, MOGP, and BLOP for 31 runs of the detection task at the uncertainty level UL=50%. The sign "+" at the ith position means that the algorithm PAURPC_d (or IAC) median value is statically different from the ith algorithm value. The sign "-" means the opposite. Similarly, the effect sizes values (small (s), medium (m), and large (l)) using the Cohen'd statistics are given. Best PAURPC_d (or IAC) values are in Bold. Second-best PAURPC_d (or IAC) values are underlined.

	Bi-ADIPOK		DECOR		GP		MOGP		BLOP	
Systems	PAURPC_d	IAC	PAURPC_d	IAC	PAURPC_d	IAC	PAURPC_d	IAC	PAURPC_d	IAC
GanttProject	**0.932** (+ + + +) (l l l l)	**0.9412** (+ + + +) (l l l l)	0.146 (- + +) (s m m)	0.1721 (+ + +) (m m m)	0.1611 (+ +) (m m)	0.1908 (- +) (s m)	0.213 (-) (s)	0.2243 (-) (s)	<u>0.224</u>	<u>0.2436</u>
ArgoUML	**0.9418** (+ + + +) (l l l l)	**0.9516** (+ + + +) (l l l l)	0.129 (- + +) (s m m)	0.1428 (- - +) (s m m)	0.145 (+ +) (m m)	0.1745 (- +) (s m)	0.201 (-) (s)	0.2125 (-) (s)	<u>0.216</u>	<u>0.2417</u>
Xercess-J	**0.9437** (+ + + +) (l l l l)	**0.9481** (+ + + +) (l l l l)	0.0805 (+ + +) (m m l)	0.0917 (+ + +) (m m l)	0.133 (+ +) (m m)	0.1609 (+ +) (m m)	0.189 (-) (s)	0.2005 (-) (s)	<u>0.203</u>	<u>0.2160</u>
JFreeChart	**0.9517** (+ + + +) (l l l l)	**0.9609** (+ + + +) (l l l l)	0.0783 (s + +) (s m l)	0.0906 (- + +) (s m l)	0.11851 (+ +) (m m)	0.1380 (+ +) (m m)	0.169 (-) (s)	0.176 (-) (s)	<u>0.176</u>	<u>0.1913</u>
Azureus	**0.9468** (+ + + +) (l l l l)	**0.9493** (+ + + +) (l l l l)	0.0465 (+ + +) (m m m)	0.0653 (- + +) (s m m)	0.1012 (+ +) (m m)	0.1211 (+ +) (m m)	0.136 (-) (s)	0.1595 (-) (s)	<u>0.1603</u>	<u>0.1682</u>
Ant-Apache	**0.9387** (+ + + +) (l l l l)	**0.9405** (+ + + +) (l l l l)	0.0429 (+ + +) (m m m)	0.0624 (- + +) (s m m)	0.0819 (+ +) (m m)	0.0988 (+ +) (m m)	0.145 (-) (s)	0.1512 (-) (s)	<u>0.15321</u>	<u>0.1637</u>

Table 4: PAURPC_d and IAC median scores of Bi-ADIPOK, DECOR, GP, MOGP, and BLOP for 31 runs of the detection task at the uncertainty level UL=0%. The sign "+" at the ith position means that the algorithm PAURPC_d (or IAC) median value is statically different from the ith algorithm value. The sign "-" means the opposite. Similarly, the effect sizes values (small (s), medium (m), and large (l)) using the Cohen'd statistics are given. Best PAURPC_d (or IAC) values are in Bold. Second-best PAURPC_d (or IAC) values are underlined.

	ADPOK		DECOR		GP		MOGP		BLOP	
Code Smells	PAURPC_d	IAC	PAURPC_d	IAC	PAURPC_d	IAC	PAURPC_d	IAC	PAURPC_d	IAC
GanttProject	**0.937** (+ + + +) (l l l l)	**0.951** (+ + + +) (l l l l)	0.351 (- + +) (s m m)	0.37815 (+ + +) (m m m)	0.401 (+ +) (m m)	0.4216 (- +) (s m)	0.531 (-) (s)	0.5574 (-) (s)	<u>0.554</u>	<u>0.5691</u>
ArgoUML	**0.9515** (+ + + +) (l l l l)	**0.9684** (+ + + +) (l l l l)	0.346 (- + +) (s m m)	0.33428 (- - +) (s s m)	0.364 (+ +) (m m)	0.3846 (- +) (s m)	0.529 (-) (s)	0.5481 (-) (s)	<u>0.5316</u>	<u>0.5659</u>
Xercess-J	**0.931** (+ + + +) (l l l l)	**0.9405** (+ + + +) (l l l l)	0.191 (+ + +) (m m l)	0.22018 (+ + +) (m m l)	0.317 (+ +) (m m)	0.3375 (- -) (s s)	0.463 (-) (s)	0.4713 (-) (s)	<u>0.505</u>	<u>0.5211</u>
JFreeChart	**0.9612** (+ + + +) (l l l l)	**0.9704** (+ + + +) (l l l l)	0.168 (- + +) (s m l)	0.20368 (- + +) (s m l)	0.296 (+ +) (m m)	0.3096 (+ +) (m m)	0.408 (-) (s)	0.4356 (-) (s)	<u>0.499</u>	<u>0.5098</u>
Azureus	**0.9503** (+ + + +) (l l l l)	**0.9617** (+ + + +) (l l l l)	0.125 (+ + +) (m m m)	0.14618 (- + +) (s m m)	0.238 (+ +) (m m)	0.2728 (- +) (s m)	0.342 (-) (s)	0.3689 (-) (s)	<u>0.416</u>	<u>0.437</u>
Ant-Apache	**0.9512** (+ + + +) (l l l l)	**0.9604** (+ + + +) (l l l l)	0.1102 (+ + +) (m m m)	0.12168 (- + +) (s m m)	0.219 (+ +) (m m)	0.2255 (- +) (s m)	0.312 (-) (s)	0.3336 (-) (s)	<u>0.337</u>	<u>0.3527</u>

Table 5: PAURPC_d and IAC median scores of Bi-ADIPOK, DECOR, GP, MOGP, and BLOP for 31 runs of the identification task at the uncertainty level L=50%. The sign "+" at the ith position means that the algorithm PAURPC_d (or IAC) median value is statically different from the ith algorithm value. The sign "-" means the opposite. Similarly, the effect sizes values (small (s), medium (m), and large (l)) using the Cohen'd statistics are given. The N/A signifies that the given approach is Not Applicable (N/A) on the corresponding smell. Best PAURPC_d (or IAC) values are in Bold. Second-best PAURPC_d (or IAC) values are underlined.

	ADIOK		DECOR		GP		MOGP		BLOP	
Code Smells	PAURPC_d	IAC	PAURPC_d	IAC	PAURPC_d	IAC	PAURPC_d	IAC	PAURPC_d	IAC
Blob	**0.9313** (+ + + +) (l l l l)	**0.9420** (+ + + +) (l l l l)	0.0764 (- + +) (s m m)	0.0927 (- + +) (s m m)	0.1543 (+ +) (m m)	0.1615 (+ +) (m m)	0.1867 (-) (s)	0.1925 (-) (s)	<u>0.2138</u>	<u>0.2190</u>
Data Class	**0.9107** (+ + +) (l l l)	**0.9258** (+ + +) (l l l)	N/A	N/A	0.1152 (+ +) (m m)	0.1243 (+ +) (m m)	0.1418 (-) (s)	0.1453 (-) (s)	<u>0.1884</u>	<u>0.1906</u>
Feature Envy	**0.8914** (+ + + +) (l l l)	**0.9412** (+ + + +) (l l l)	N/A	N/A	0.1089 (+ +) (m m)	0.1195 (+ +) (m m)	0.1402 (-) (s)	0.1451 (-) (s)	<u>0.1809</u>	<u>0.1821</u>
Long Method	**0.8820** (+ + +) (l l l)	**0.8943** (+ + +) (l l l)	N/A	N/A	0.1013 (+ +) (m m)	0.1107 (+ +) (m m)	0.1349 (-) (s)	0.1363 (-) (s)	<u>0.1479</u>	<u>0.1528</u>
Duplicate Code	**0.8820** (+ + +) (l l l)	**0.8951** (+ + +) (l l l)	N/A	N/A	0.0768 (+ +) (m l)	0.0890 (+ +) (m l)	0.1074 (+) (m)	0.1124 (+) (m)	<u>0.1267</u>	<u>0.1348</u>
Long Parameter List	**0.8914** (+ + +) (l l l)	**0.9063** (+ + +) (l l l)	N/A	N/A	0.0691 (+ +) (l l)	0.0724 (+ +) (l l)	0.1031 (-) (s)	0.1072 (-) (s)	<u>0.1181</u>	<u>0.1201</u>
Spaghetti Code	**0.8853** (+ + + +) (l l l l)	**0.8847** (+ + + +) (l l l l)	0.0328 (+ + +) (l l l)	0.0447 (+ + +) (l l l)	0.0544 (+ +) (m l)	0.0625 (+ +) (m l)	0.0798 (-) (s)	0.0851 (-) (s)	<u>0.1004</u>	<u>0.1030</u>
Functional Decomposition	**0.8612** (+ + + +) (l l l l)	**0.8805** (+ + + +) (l l l l)	0.0213 (+ + +) (l l l)	0.0335 (+ + +) (l l l)	0.0495 (+ +) (m l)	0.0584 (- +) (s l)	0.0607 (-) (s)	0.0615 (-) (s)	<u>0.0682</u>	<u>0.0716</u>

Table 6: PAURPC_d and IAC median scores of Bi-ADIPOK, DECOR, GP, MOGP, and BLOP for 31 runs of the identification task at the uncertainty level L=0%. The sign "+" at the ith position means that the algorithm PAURPC_d (or IAC) median value is statically different from the ith algorithm value. The sign "-" means the opposite. Similarly, the effect sizes values (small (s), medium (m), and large (l)) using the Cohen'd statistics are given. The N/A signifies that the given approach is Not Applicable (N/A) on the corresponding smell. Best PAURPC_d (or IAC) values are in Bold. Second-best PAURPC_d (or IAC) values are underlined.

	Bi-ADIPOK		DECOR		GP		MOGP		BLOP	
Code Smells	PAURPC_d	IAC	PAURPC_d	IAC	PAURPC_d	IAC	PAURPC_d	IAC	PAURPC_d	IAC
Blob	**0.9469** (+ + + +) (l l l l)	**0.9519** (+ + + +) (l l l l)	0.3482 (- + +) (s m m)	0.3520 (- + +) (s m m)	0.3379 (+ +) (m m)	0.3484 (+ +) (m m)	0.4211 (-) (s)	0.4239 (-) (s)	<u>0.4482</u>	<u>0.4507</u>
Data Class	**0.9312** (+ + +) (l l l)	**0.9214** (+ + +) (l l l)	N/A	N/A	0.2756 (+ +) (m m)	0.2772 (+ +) (m m)	0.3204 (-) (s)	0.326 (-) (s)	<u>0.3726</u>	<u>0.3819</u>
Feature Envy	**0.8930** (+ + +) (l l l)	**0.9015** (+ + +) (l l l)	N/A	N/A	0.2642 (+ +) (m m)	0.2691 (+ +) (m m)	0.3269 (-) (s)	0.3321 (-) (s)	<u>0.3810</u>	<u>0.395</u>
Long Method	**0.8976** (+ + +) (l l l)	**0.8905** (+ + +) (l l l)	N/A	N/A	0.2484 (+ +) (l l)	0.253 (+ +) (l l)	0.3094 (-) (s)	0.3180 (-) (s)	<u>0.3709</u>	<u>0.3782</u>
Duplicate Code	**0.8843** (+ + +) (l l l)	**0.8815** (+ + +) (l l l)	N/A	N/A	0.2340 (+ +) (m l)	0.2367 (+ +) (m l)	0.2770 (-) (s)	0.2891 (-) (s)	<u>0.3172</u>	<u>0.3296</u>
Long Parameter List	**0.8914** (+ + +) (l l l)	**0.9022** (+ + +) (l l l)	N/A	N/A	0.1870 (+ +) (l l)	0.1902 (+ +) (l l)	0.2468 (+) (m)	0.253 (+) (m)	<u>0.2914</u>	<u>0.3011</u>
Spaghetti Code	**0.8752** (+ + + +) (l l l l)	**0.8817** (+ + + +) (l l l l)	0.1082 (- + +) (s l l)	0.117 (- + +) (s l l)	0.1244 (+ +) (l l)	0.1299 (+ +) (l l)	0.2019 (+) (m)	0.204 (+) (m)	<u>0.2507</u>	<u>0.2640</u>
Functional Decomposition	**0.8815** (+ + + +) (l l l l)	**0.8916** (+ + + +) (l l l l)	0.0825 (+ + +) (m l l)	0.0883 (+ + +) (m l l)	0.1180 (+ +) (m l)	0.1251 (+ +) (m l)	0.1423 (+) (m)	0.161 (+) (m)	<u>0.1966</u>	<u>0.2093</u>

ones, leading to a significant imbalance ratio within the base of examples. In the case of an uncertain environment, the software class labels are represented in the form of class labels, i.e., each software class label has been defined by a vector of two real numbers. Each vector value is a degree of possibility that indicates the degree of membership of a particular software class (instance) to each smell and Non-smelly class labels. Table 5 shows the PAURPC_d and IAC values of Bi-ADIPOK, MOGP, GP, DECOR, and BLOP. We point out from this table that our Bi-ADIPOK approach is superior to the remaining considered approaches in terms of the used performance metrics. For Bi-ADIPOK, the PAURPC_d values vary between [0.8576, 0.9273] and the IAC values lie between [0.8693, 0.9318]. However, the BLOP approach ranks second as its PAURPC_d values are between [0.0682, 0.2138] and its AUC values lie between [0.0716, 0.2190]. The other approaches, that is to say, DECOR, GP, and MOGP, achieved poorer results with maximums of 0.0764, 0.1543, 0.1867, and 0.0927, 0.1615, 0.1925 for the PAURPC_d and IAC metrics, respectively. These outcomes could be explained by the following two reasons. On the one side, the data imbalance ratio is high through the identification process and all the existing approaches are inappropriate for this problem. On the other side, the identification is carried out in an uncertain setting. Therefore, the imbalance problem can not be dealt with their adopted detectors' structures as well as their fitness functions. Table 5 shows the statistical results for the identification case under uncertain environment. These results proves that our approach (i.e., Bi-ADIPOK) is more significant than DÉCOR, GP, MOGP, and BLOP. Such outperformance could be explained by the fact that our proposed approach is able to handle the problem of uncertain developers' opinion thanks to the adopted fitness function.

In the case of a certain environment, the whole class labels inherent in the BE are certain. Based on Table 6, the Bi-ADIPOK approach surpasses all the remaining considered approaches with PAURPC_d varying between 0.8656 and 0.9351. The second-best approach belongs to the BLOP approach where its PAURPC_d lies between 0.1966 and 0.4482. The remaining approaches are ranked (based on PAURPC_d values) in the following order (from the poorest

to the best): GP (0.3379), DECOR (0.3482), and MOGP (0.4211). For the IAC metric, similar outcomes are obtained. These outcomes are clarified by the fact that the employed fitness functions enable us to get suitable detectors for imbalanced data whereas the other approaches have got bad detectors as the fitness function of these detectors are inappropriate to address data imbalance issue. To sum up, the degradation in the overall performance did not affect all the approaches of the same size. The Bi-ADIPOK approach shows a slightly lower quality of the results compared to the one demonstrated during the detection process. In contrast, the performance metrics' of DECOR, GP, BLOP, MOGP are considerably reduced. It is possible to clarify these findings as follows. In the case of the DECOR approach, the generated results were dire poor for the reason that the employed rules are predefined. Such a fact will not render DECOR suitable to address the data imbalance issue as well as the uncertain class labels one. However, the results of GP, MOGP, and BLOP, vary between poor and very poor since their identification process has been evolved based on the metaheuristic algorithms. For this reason, these approaches can discover some smell types by accident. Generally, we can deduce from the various obtained results that all the approaches were not similarly affected by the deterioration of the performance. Our Bi-ADIPOK approach succeeds to achieve better results in the process of detection as well as identification one. Nevertheless, when treating both processes, the considered approaches (excluding ours) have demonstrated their deeper weaknesses. For the DECOR approach, since its detection rules are predefined, hence its findings were extremely poor. In contrast, the search-based approaches used in the experimental study, their results vary between poor and very poor. This fact could be explained by the fact that their identification processes have been evolved based on metaheuristic algorithms. Therefore, the detected smells are made by chance. The statistical results for the identification case under certain environment shown in Table 6, proves that Bi-ADIPOK approach is more significant than the remaining approaches (i.e., DÉCOR, GP, MOGP, and BLOP) and this could be explained by the fact that our approach is able to deal with the problem of imbalanced data.

5.6.3. Results for RQ3

To reply to the RQ3, we would like to point out that the principal objective of this research question is to assess the overall performance of our Bi-ADIPOK against the baseline PK-NN. For the comparison, we adopted the PK-NN since it is able to manage the uncertainty inherent in the BE class labels over the building process. However, the PK-NN relies on the greedy search strategy to build its classifiers. More precisely, the greedy search is performed to fix the PK-NN' parameters. Therefore, the two approaches have been compared under two environments, i.e., uncertain and certain, for the detection ad identification tasks.

For the case of the detection in an uncertain environment (cf., Table 7), the PAURPC_d metric values of our ADPOK approach lie between 0.902 and 0.932, while the IAC metric values vary between 0.9108 and 09407. In contrast, the PK-NN records 0.3684 and 0.5013 for the PAURPC_d metric, 0.3703 and 0.5215 for the IAC metric. Similarly, the Bi-ADIPOK approach largely exceeds the baseline PK-NN for the detection under certain environment, the Bi-ADIPOK PAURPC_d values belong to [0.928, 0.957] whereas those of the PK-NN belong to [0.3359, 0.5162]. For the IAC metric, the same observation is valid.

For the case of identification in an uncertain environment (cf., Table 8), Bi-ADIPOK surpasses the PK-NN for the identification of eight types of code smell under both environments: certain and uncertain. These obtained results could be clarified as follows. On the one side, through the adopted GA, the Bi-ADIPOK approach can prevent falling into local optima as well as getting close to the global optima. However, the PK-NN approach conducts a greedy search through the search space that will push it to get stack into the local optima. On the other side, the Bi-ADIPOK used an appropriate fitness function that can deal with the data imbalance problem in addition to the uncertain class labels. However, the PK-NN is almost able to handle the uncertainty. For this reason, the baseline approach has shown a medium quality during the detection process as this latter is characterized by a lower ratio of data imbalance. The

Bi-ADIPOK approach produces a set of optimized detectors for every considered smell type. Contrary, the baseline PK-NN generates only non-optimized detectors. It is important to note that the results deterioration of Bi-ADIPOK through the identification process could be clarified by the significantly higher ratio of the data imbalance. In contrast, the PK-NN suffers from a remarkable degradation. Table 7 and 8 show the statistical results of Bi-ADIPOK and baseline PK-NN in the detection and identification under uncertain and certain environments. The obtained results clearly demonstrate that Bi-ADIPOK approach is more significant that the baseline PK-NN

6. Threats to validity

The various factors that could distort our empirical research are discussed in this section. These factors could be split into three different categories that are: (1) internal, (2) external, and (3) construct validity. The first category, internal validity threats, is related to the correctness of the obtained outcomes of our proposal's experiments, whereas the second category (i.e., external validity threats) concerns the generalization of the obtained outcomes. The last category of threats, i.e., construct validity, concerns the theory-observation link.

6.1. Threats to internal validity

In this work, we took into account the internal threats to validity in the stochastic algorithm use since our experiments are based on 31 independent simulations for every instance of the problem, and the outcomes are assessed using the Wilcoxon-rank sum test with a confidence level equals to 95% and the alpha equals to 5%. During this work, the configuration of the parameter related to the different optimization algorithms used in our work activates a crucial internal threat that needs to be addressed in our forthcoming work. Furthermore, the calibration of the parameters in the experimental study is performed through the commonly employed technique in the SBSE community known as the trial-and-error technique [64, 70]. To handle such a threat, it

Table 7: PAURPC_d and IAC median scores of Bi-ADIPOK and PK-NN baseline approach over 31 independent runs regarding the detection task the uncertainty levels UL=50% and UL=0%. The sign "+" at the ith location means that the algorithm PAURPC_d (or IAC) median score is statically different from the ith algorithm score. The sign "-" means the opposite. Similarly, the effect sizes scores (small (s), medium (m), and large (l)) using the Cohen'd statistics are given. Best PAURPC_d (or IAC) scores are in Bold.

Systems	UL	Bi-ADIPOK		baseline PK-NN	
		PAURPC_d	IAC	PAURPC_d	IAC
GanttProject	UL= 50 %	0.902 (+) (l)	0.9108 (+) (l)	0.5013	0.5215
	UL= 0%	0.928 (+) (l)	0.9477 (+) (l)	0.5162	0.5329
ArgoUML	UL= 50 %	0.9227 (+) (l)	0.9246 (+) (l)	0.4927	0.5068
	UL= 0%	0.943 (+) (l)	0.9597 (+) (l)	0.503	0.5315
Xercess-J	UL= 50 %	0.914 (+) (l)	0.9130 (+) (l)	0.4815	0.4920
	UL= 0%	0.931 (+) (l)	0.9405 (+) (l)	0.4620	0.4721
JFreeChart	UL= 50 %	0.932 (+) (l)	0.9407 (+) (l)	0.4009	0.4253
	UL= 0%	0.957 (+) (l)	0.9622 (+) (l)	0.3855	0.4062
Azureus	UL= 50 %	0.9307 (+) (l)	0.9356 (+) (l)	0.3817	0.3941
	UL= 0%	0.946 (+) (l)	0.9593 (+) (l)	0.3482	0.3605
Ant-Apache	UL= 50 %	0.923 (+) (l)	0.9293 (+) (l)	0.3684	0.3703
	UL= 0%	0.9423 (+) (l)	0.955 (+) (l)	0.3359	0.3482

Table 8: PAURPC_d and IAC median scores of Bi-ADIPOK and PK-NN baseline approach over 31 independent runs regarding the identification task the uncertainty levels UL = 50% and UL=0%. The sign "+" at the ith location means that the algorithm PAURPC_d (or IAC) median score is statically different from the ith algorithm score. The sign "-" means the opposite. Similarly, the effect sizes scores (small (s), medium (m), and large (l)) using the Cohen'd statistics are given. Best PAURPC_d (or IAC) scores are in Bold.

Systems	UL	Bi-ADIPOK		baseline PK-NN	
		PAURPC_d	IAC	PAURPC_d	IAC
Blob	UL= 50 %	0.9273 (+) (l)	0.9318 (+) (l)	0.3706	0.384
	UL= 0%	0.9351 (+) (l)	0.9476 (+) (l)	0.4015	0.4124
Data Class	UL= 50 %	0.9054 (+) (l)	0.9133 (+) (l)	0.0.3689	0.3762
	UL= 0%	0.9012 (+) (l)	0.9154 (+) (l)	0.3170	0.3207
Feature Envy	UL= 50 %	0.8872 (+) (l)	0.8962 (+) (l)	0.3154	0.3288
	UL= 0%	0.8874 (+) (l)	0.8901 (+) (l)	0.3290	0.3302
Long Method	UL= 50 %	0.8713 (+) (l)	0.8835 (+) (l)	0.3062	0.3109
	UL= 0%	0.8803 (+) (l)	0.8820 (+) (l)	0.3158	0.3177
Duplicate Code	UL= 50 %	0.8702 (+) (l)	0.8821 (+) (l)	0.2976	0.3082
	UL= 0%	0.8694 (+) (l)	0.8715 (+) (l)	0.2813	0.2957
Long Parameter List	UL= 50 %	0.8835 (+) (l)	0.8976 (+) (l)	0.2910	0.2976
	UL= 0%	0.8775 (+) (l)	0.8844 (+) (l)	0.2387	0.2492
Spaghetti Code	UL= 50 %	0.8726 (+) (l)	0.8741 (+) (l)	0.239	0.2413
	UL= 0%	0.8613 (+) (l)	0.8672 (+) (l)	0.2146	0.2301
Functional Decomposition	UL= 50 %	0.8576 (+) (l)	0.8693 (+) (l)	0.1985	0.2110
	UL= 0%	0.8652 (+) (l)	0.8703 (+) (l)	0.1502	0.1589

could be an interesting prospect if we realize a configuration strategy that aims to update our approach parameters until the best ones have been achieved.

6.2. Threats to external validity

The external threat to validity principally discusses the considered smell types in addition to the set of software systems studied through this study. We have studied eight code smell types (cf. Appendix 9), which are a widely representative and most frequent collection of smell types. Similarly, we have selected from different application domains, six different software systems (cf. Table 1) with various sizes as well as varied functionalities, and further developed by diverse companies. The purpose of choosing such different systems is to decrease the bias that may occur as a result of the specific character on the picked systems. To mitigate the bias evoked by the chosen systems, we conducted a K-folds cross-validation strategy. It is also of great interest to test our proposed Bi-ADIPOK tool to identify anti-patterns residing in web services and Android applications. For the web-services applications, the existing anti-patterns may hinder their processing and thus decrease their quality as well as their rate of utilization. Concerning the Android applications, the residing anti-patterns could negatively influence the execution of these applications by reducing the processing time and by rising the energy consumption.

6.3. Threats to construct validity

During our experimental study, we have constructed a BE for the detection of anti-patterns by means of some known advisors (cf. Figure 3) that are: JDeodorant [50], DECOR [49], iPlasma[9], inFusion[10], and PMD [51]. The generated BE is injected with uncertainty, precisely, at the level of class labels. The conversion from crisp (or certain) class labels to the uncertain ones are performed through five distinct types of probabilistic classifiers (Naïve Bayes classifier [41], Probabilistic K-NN [42], Bayesian Networks [43, 44], Naïve Bayes

[9] http://loose.upt.ro/iplasma/
[10] http://www.intooitus.com/products/infusion

Nearest Neighbor [45], and Probabilistic Decision Tree [46]) with the aim to create probability values for the existing class labels. Then, we used an existing mathematical formula for the conversion of probability distributions into possibility ones (more details are given in Section 4.1). Thus, a constructed threat to validity might be linked to the use of the set of chosen probabilistic classifiers that aims to simulate the subjectivity and the uncertainty of the experts' opinions through the process of likelihood values generation. To deal with such issue, the obtained values will be manually verified by human experts. Our research study is the first work in the SBSE field that detects and identifies code smells under uncertainty that resides at the level of class labels. Accordingly, a crucial construct threat to validity occurs since there is no SBSE work dealing with the detection of code smells under uncertain environment. Existing approaches neglect the existing uncertainty in the BE. For the comparison, we assessed our approach with a baseline (possibilistic) one (i.e., PK-NN). In fact, we re-implemented the baseline approach since it is not available. Thus, the PK-NN could be wrongly re-implemented and consequently the obtained results could be skewed. To diminish this threat, we based on experienced code reviewers to ensure effective implementation. Moreover, we compared the outcomes of the re-implemented PK-NN presented in the literature, so the comparison demonstrates that the outcomes are almost identical.

7. Related works

The code smell detection continues to be an extremely active and opportune subject for research in the SE field, in particular the SBSE one [72]. Different authors have suggested several research types for automating the detection methods of code smells to assist the developers in the detection task. In a number of SE problems [73], the term uncertainty has emerged, in which the authors the researchers have addressed the uncertainty provoked by changing environmental conditions by varying environmental conditions in the case of self-adapting systems. We notice that the uncertainty was also mentioned by [74]

in an attempt to optimize the non-functional requirements of the self-adaptive system, which produces adjustment strategies for making reconfigurations at run time to deal with issues that are unforeseen due to uncertainty like unpredictable issues in the system itself. However, the majority of works that have been conducted in the SE have not dealt with real imperfect data, including the SBSE researches that can be categorized into four main groups: (1) Rule-based methods, (2) Machine learning-based methods (coupled with deep learning), (3) Search-based methods, and (4) Others.

7.1. Rule/heuristic-based approaches

The initial attempts to identify the software classes, which contain anti-patterns focused on determining rule-based methods (also known as heuristic-based approaches) [75] that depend on structural metrics to capture detours from best practices of object-oriented design. The first study was conducted by Erni and Lewerentz [76] that employed quality metrics to assess the performance of the framework in order to enhance it. The authors employed the multi-metric definition, whereby the m-tuple of various metrics determines a quality criterion. Such was Marinescu [77]'s proposal to use a metric-based approach to analyze the code source for the identification of defects on various levels of object-oriented design fragments (along with method, class, and subsystem). In this respect, Lanza and Marinescu [78]'s detection strategy consisted of a combination of structural metrics with thresholds several detection rules for 11 anti-patterns were laid down. These rules consist of several metric threshold pairs connected by the operators of AND/OR. Such heuristics have been implemented within the InCode tool [79]. Furthermore, Moha et al. [49] proposed in another research the DECOR approach, which comprises the major steps in the specification and detection of code smells. More precisely, this approach begins by describing the symptoms of defects through abstract rules language. These descriptions contain various concepts, including class roles and structures, mapped to an algorithm for detection. More recently, the clustering methods used to detect code smells have been adopted. On the basis of

this consideration, Tsantalis and Chatzigeorgiou [50] suggested the JDeodorant tool for detecting code smells and recommending certain move methods as a refactoring procedure. The tool is first established to detect only the Feature Envy smell and then upgraded to deal with some other smell types like the blob, state checking, and long method [80, 81]. The detection mechanism for JDeodorant relies on a set of code metrics that are linked to the cut-off of the resulted dendrograms using the clustering techniques as well as thresholds.

7.2. Machine Learning-based approaches

A new trend has recently been raised, which comprises the use of machine learning techniques for code smell detection. In this portion, the adopted techniques principally belong to the supervised techniques. In particular, these techniques are constructed based on training data and then carried out on software systems to predict the smelly software classes. Initially, Kreimer [82] suggested a model that merges known methods to detect the occurrences of design defects like God class and Long Method on the basis of code metrics as attributes to build Decision Trees (DTs). Two small software systems, IYC and WEKA, were evaluated. After ten years, previous findings were confirmed by Amorim et al. [83], which evaluated the performance of DTs classifiers on different software projects to recognize 12 anti-patterns.

Khomh et al. [84] suggested that Bayesian Networks be used to detect Blob code smell occurring in open source software systems (GanttProject, Xerces-J). Subsequently, Khomh et al. [85] expanded their work to a novel one called Bayesian Detection Expert (BDTEX). This latter was validated on different smell types like God Class, Functional Decomposition, and Spaghetti Code. The BDTEX approach employs the Goal Question Metric for the construction of the Belief Bayesian Network (BBN) to infer information from the definition of code smells. The BDTEX detection outputs are probability (rather than Boolean) values that are attributed to the code component containing code smell. Khomh et al. [85] also employed the BBNs to investigate and the lifecycle of the BLOB' evolution from those who have accidentally occurred (i.e., bad code).

Maiga et al. [86, 87] introduced the SVMDetect method, which identifies anti-patterns with the Support Vector Machine (SVM) technique. This proposed approach has been approved with four types of anti-patterns (like Blob, Functional Decomposition, Spaghetti Code, and Swiss Army Knife) upon ArgoUML, Azureus, and Xerces software projects. The authors then broadened their approach to a more effective one named SMURF, which tooks into account the various participants' feedbacks.

Hassaine et al. [88] adopted an immune-inspired approach for the detection of BLOB smell type. The designed approach detects the smells in software classes, especially those that violate the features of certain rules. Likewise, Oliveto et al. [89] presented an approach named Anti-pattern identification using B-Splins (ABS) that identifies smelly instances via a technique for numerical analysis.

Recently, the authors have investigated the performance of diverse ML techniques for the problem of code smells detection. Additionally, Fontana et al. [90] surveyed 16 supervised ML methods with their boost variant to detect anti-patterns, like Blob, Data Class, Feature Envy, and Long Method, on 74 software projects. Furthermore, over the training and the assessment phases, the authors adopted a technique of under-sampling in the interest of poor performance of some ML techniques when datasets were imbalanced. In another research, Fontana and Zanoni [91] made a classification of code smells severity on the basis of a regression technique and multinomial classification. Such an approach may help developers to prioritize or to rank classes or methods. However, Di Nucci et al. [20] referenced that Fontana et al. [90] had limited the way the dataset would be constructed. Accordingly, the authors set up Fontana's datasets and create new ones, which are suitable for real-world scenarios.

7.3. Search-based approaches

In software engineering, search-based approaches are adopted to alleviate diverse optimization problems with meta-heuristic techniques like *GA*, GP, and so on. The detection stage is seen as the main stage since over this stage the

existing smell types, as well as their paths in the software systems, are reported before the refactoring (correction) phase started. For instance, the departure point was with Kessentini et al. [13] where they suggested an automated solution based on detection rules to identify the existing code smells in a particular software project. These detection rules are expressed as combinations of quality metrics together with thresholds derived from the comparison of diverse heuristic search algorithms for rules extraction such as Simulated Annealing, Harmony Search, and Particle Swarm Optimization. The experiments were conducted on a series of anti-patterns (Spaghetti Code, God Class, and Functional Decomposition) on the basis of Xerces-J and GanttProject Software projects. Then, Ouni et al. [61] have implemented a search-based approach to detect current smells in software projects. This proposed approach was the first to infer detection rules from the smelly examples through Genetic Programming. For the experimentation study, the authors have applied their approach on Blob, Spaghetti Code, and Functional Decomposition anti-patterns types relying on a variety of software projects with diverse scales as GanttProject, Xerces-J, ArgoUML, Quick UML, LOG4J, and AZUREUS.

Boussaa et al. [92] have conceived an approach, which leans on a competitive co-evolutionary search to tackle the problem of code smell detection. Over this proposed approach, two competing populations simultaneously evolve where the first population is devoted to generate a number of detection rules aiming to enhance the ratio of the detection task of smelly examples, while the second population tries to increase the amount of artificially build code smells, which the first population does not detect. The experimental study was carried out on the basis of a number of software projects having various sizes, namely Xerces, Azureus, Ant-Apache, and ArgoUML aiming to detect three types of code smells viz., Spaghetti Code, Functional Decomposition, and Blob.

Kessentini et al. [93] introduced the parallel way to their conceived approach namely the Parallel Evolutionary Algorithm (PEA) aiming to detect code smells. This approach the Genetic Algorithm and Genetic Programming simultaneously across the optimization stage to generate a number of code smell detection

rules based on various examples of code smells as well as well-designed (non-smelly) code fragments examples respectively. The PEA approach has been tested on distinct types of code smells (such as spaghetti code, feature envy, data class, lazy class, Blob, functional decomposition, shotgun surgery, and long parameter list) based upon open sources software systems like ApacheAnt V1.5.2, ApacheAnt V1.7.0, Nutch, Log4J, Lucene, Xerces-J, Rhino, JFreechart, and Ganttproject.

Sahin et al. [40] have proposed an approach called Bi-Level Optimization Problem (BLOP) approach that uses the bi-level optimization technique to generate rules for code smell detection relying on two levels: (1) the upper and (2) the lower levels. The former is involved in the creation of a range of code smell detection rules aiming to raise the coverage of not only the code smell examples but also the artificial code smells obtained from the lower level. The latter has the responsibility to disclose the most potential smells from the upper level that are not recognized by the obtained detection rules. The BLOP approach was tested on a variety of code smells (like Long Parameter List, Functional Decomposition, Blob, Feature Envy, Data Class, Spaghetti Code, and Lazy Class) and on the basis of nine software systems having large and medium sizes viz., JFreeChart, GanttProject, ApacheAnt, Nutch, Log4J, Lucene, Xerces-J, and Rhino.

Mansoor et al. [18] have suggested a method that relies on the multi-objective optimization strategy, namely Multi-objective Genetic Programming (MOGP) for the task of generating the range of detection rules. This approach has been employed to identify the best combination of quality metrics that raises the number of the detected code smell examples while simultaneously decreases the number of detected well-conceived examples. The MOGP approach has been evaluated on various code smells (Blob, Feature Envy, Data Class, Spaghetti Code, Functional Decomposition) using diverse Object-Oriented Software projects: ArgoUML v0.26, ArgoUML v0.3, Xerces-J, Ant-Apache v1.5, Ant-Apache v1.7.0, GanttProject, Azureus.

7.4. Others

Recently, a number of researchers addressed the code smell detection problem from the historical information viewpoint for the code source evolution. This concept initially begins with Rapu et al. [94], which inferred the historical information from the defective code source structure. The study analyzed a collection of historical measurements that reflect the evolution of code smells. The findings obtained were combined with the initial strategies for detection. On the basis of three open-source projects, two in-house projects, and Jan (3D graphics environment), the researchers assessed their proposed approach upon the identification of two smell types: Blob, Data Class. In another study, an approach namely Historical Information for Smell detection (HIST) was developed by Palomba et al. [95, 96]. The proposed approach involves the detection of code smells through the projects' historical information that are inferred from the revision control system. Five distinct types of code smells (Feature Envy, Divergent Change, Blob, Shotgun Surgery, and Parallel Inheritance) in addition to a set of software projects were adopted in the experimental study of the HIST work approach. In fact, the authors have raised the number of software projects to 20 in the latest work. Likewise, Fu and Shen [97] suggested a tool (founded on the associated rule mining) that is capable of deriving data history (concerned with addition or modification, either methods or classes or packages) from projects with an enormous growth history length. The authors seek to validate their proposed method by identifying three code smells, namely shotgun surgery, duplicate code, and divergent change, using five open source systems that are: Eclipse, Closure Compiler, jUnit, Guava, and Maven.

Some researchers like Emden and Moonen [98] have focused on visualizing anti-patterns for complicated software analysis like the JCOSMO tool. This proposed approach involves parsing the Java source code and using the graphical view to display the defective code fragments by smells and their links. Subsequently, the VERSO framework in which the process of visualization relied on colors that represent properties and are mainly intended to facilitate software quality analysis. Such tool has been suggested by Langelier et al. [99]. Similar

to the same features of the previous tool, another tool for code smell detection has been proposed by Dhambri et al. [100] in which a part of the detection is performed automatically while the other part is left to the human analyst for the judgment. Such a tool was tested various design anomalies (viz., Blob, Functional Decomposition, and Divergent Change) using two software projects such as PMD and Xerces-J.

8. Conclusion and future works

In this paper, we proposed Bi-ADIPOK as a new approach for detecting and identifying code smells under certain and uncertain environments where the uncertainty occurs at the level of class labels. Through this original research study, we launched an original trend in the Software Engineering field and more specifically in the SBSE one, which corresponds to the consideration of code smell detection and identification as an uncertain classification problem. The uncertainty sources could be: (1) the subjectivity and/or doubtfulness of software engineers about the system classes smelliness or (2) diverse opinions belonging to the human experts regarding the types of the existing smells inside the processed classes of a given software project. Taking decisions under uncertainty generally leads to biased results. Unfortunately, existing works in the SBSE community usually overlook the issue of uncertain class labels and choose a single possible code smell type from various ones. Ignoring and/or discarding the uncertainty lead to the deterioration of the detectors' performances. Moreover, the good results shown by most existing detectors are biased since they suffer from the inability to deal with uncertainty. To mitigate the uncertainty problem, we have proposed the Bi-ADIPOK approach, which evolves a set of PK-NNs classifiers based on the optimization of the PAURPC_d in which the possibility distribution at each software class label is taken into account. Such a measure is not only adequate for the case of uncertain classification problems but also suitable for the case of imbalanced data classification issue. The obtained outcomes prove the merits of our developed approach against the four

relevant state-of-the-art approaches and the PK-NN baseline approach. The surpass of our Bi-ADIPOK approach might be justified based on the following reasons. First, it adopts a suitable scripter (PK-NN classifier) that is adequate to handle the issue of uncertainty residing over the class labels. Such a chosen classifier is evolved via the employment of the GA in order to avoid falling into local optima and simultaneously approaching the global optima. Second, the developed fitness function is able to manage simultaneously the uncertain and imbalanced data. Third, Bi-ADIPOK employs the possibility theory tool to manage the uncertainty existing at the level of software class labels based on the possibility distributions. Finally, based on the solutions offered by the possibility theory, our proposed approach adopts the fusion operator namely Adaptive Fusion Operator (AFO) for merging the various possibility distributions generated by several detectors. Thanks to the AFO tool, our Bi-ADIPOK becomes able to merge non-conflictual and conflictual information, more precisely possibility distributions, simultaneously.

As future works, we intend to broaden our BE by generating artificial code smell types that are not covered throughout this work. Another issue can be faced by the researchers which corresponds to the fact that a considerable amount of data could be unlabeled. This issue could be mitigated based on one of the semi-supervised techniques as an amount of labeled data exist in addition to a huge amount of unlabeled data. Handling such issue could bring interesting results to the SBSE community in addition to the uncertain classification problem. Moreover, we aim to fusion different types of data (like historical and structural ones) using adequate classifiers for the learning process in order to well detect and/or identify the present code smells especially under uncertain environment. In addition, in this work, we have used the possibility theory to simulate the subjectivity and doubtfoulness of the software engineers regarding the smelliness of the software classes. However, it will be interesting to use other uncertainty theories such as the imprecise probability, evidence theory, etc [101, 102]. Finally, in our work, we have investigated the performance of Bi-ADIPOK approach on unseen software classes obtained from the six software

projects used in the training phase. However, we have not predicted the labels of the target software projects [103, 104]). Hence, equipping our Bi-ADIPOK with a transfer learner capable of transferring knowledge from the considered source projects to target ones from various domains will be interesting [105, 106, 107].

Acknowledgements

Carlos A. Coello Coello gratefully acknowledges support from CONACyT grant no. 2016-01-1920 (Investigación en Fronteras de la Ciencia 2016). He was also partially supported by the Basque Government through the BERC 2022-2025 program and by Spanish Ministry of Economy and Competitiveness MINECO: BCAM Severo Ochoa excellence accreditation SEV-2017-0718.

References

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

[2] B. Alkhazi, A. DiStasi, W. Aljedaani, H. Alrubaye, X. Ye, M. W. Mkaouer, Learning to rank developers for bug report assignment, Applied Soft Computing 95 (2020) 1–15.

[3] J. Hernández-González, D. Rodriguez, I. Inza, R. Harrison, J. A. Lozano, Learning to classify software defects from crowds: a novel approach, Applied Soft Computing 62 (2018) 1–29.

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

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

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

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

[8] M. I. Azeem, F. Palomba, L. Shi, Q. Wang, Machine learning techniques for code smell detection: A systematic literature review and meta-analysis, Information and Software Technology 108 (2019) 115–138.

[9] G. Catolino, F. Palomba, F. A. Fontana, A. De Lucia, A. Zaidman, F. Ferrucci, Improving change prediction models with code smell-related information, Empirical Software Engineering 25 (1) (2020) 49–95.

[10] H. Tong, C. Zhang, F. Wang, Code smell detection based on multi-dimensional software data and complex networks, in: International Conference of Pioneering Computer Scientists, Engineers and Educators, Springer, 2020, pp. 490–505.

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

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

[13] M. Kessentini, H. Sahraoui, M. Boukadoum, M. Wimmer, Search-Based Design Defects Detection by Example, in: Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering,, Vol. 6603, Springer, 2011, pp. 401–415.

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

[15] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, A. De Lucia, The scent of a smell: An extensive comparison between textual and structural smells, IEEE Transactions on Software Engineering 44 (10) (2017) 977–1000.

[16] R. C. Barros, M. P. Basgalupp, A. C. De Carvalho, A. A. Freitas, A survey of evolutionary algorithms for decision-tree induction, IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews) 42 (3) (2012) 291–312.

[17] H. Al-Sahaf, Y. Bi, Q. Chen, A. Lensen, Y. Mei, Y. Sun, B. Tran, B. Xue, M. Zhang, A survey on evolutionary machine learning, Journal of the Royal Society of New Zealand 49 (2) (2019) 205–228.

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

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

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

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

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

[23] M. V. Mantyla, J. Vanhanen, C. Lassenius, Bad smells-humans as code critics, in: 20th IEEE International Conference on Software Maintenance, 2004. Proceedings., IEEE, 2004, pp. 399–408.

[24] A. S. Foundation, Apache commons cli, [Accessed 19-April-2021] (2004). URL `http://commons.apache.org/cli/`

[25] I. Jenhani, S. Benferhat, Z. Elouedi, 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, 2009, pp. 505–517.

[26] I. Jenhani, N. B. Amor, Z. Elouedi, Decision trees as possibilistic classifiers, International Journal of Approximate Reasoning 48 (3) (2008) 784–807.

[27] S. Tsang, B. Kao, K. Y. Yip, W.-S. Ho, S. D. Lee, Decision trees for uncertain data, IEEE transactions on knowledge and data engineering 23 (1) (2009) 64–78.

[28] S. Saied, Z. Elouedi, K-nearest neighbors classifier under possibility framework, in: Proceedings of the 27th La Logique Floue est ses Applications, LFA, 2018, pp. 1–8.

[29] D. Dubois, H. Prade, La fusion d'informations imprécises, Traitement du signal 11 (6) (1994) 447–458.

[30] L. A. Zadeh, Fuzzy sets as a basis for a theory of possibility, Fuzzy sets and systems 1 (1) (1978) 3–28.

[31] D. Dubois, H. Prade, Possibility Theory: An Approach to Computerized Processing of Uncertainty, Plenum Press, New York, 1988.

[32] D. Dubois, H. Prade, Possibility theory in information fusion, in: Proceedings of the 3rd international conference on information fusion, Vol. 1, IEEE, 2000, pp. 6–19.

[33] N. Dunford, J. Schwartz, B. WG, B. RG, Linear Operators, Wiley-Interscience, New York, 1971.

[34] M. Higashi, G. J. Klir, On the notion of distance representing information closeness: Possibility and probability distributions, International Journal of General System 9 (2) (1983) 103–115.

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

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

[37] I. Jenhani, N. B. Amor, Z. Elouedi, S. Benferhat, K. Mellouli, 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, 2007, pp. 840–852.

[38] B. Colson, P. Marcotte, G. Savard, An overview of bilevel optimization, Annals of operations research 153 (1) (2007) 235–256.

[39] S. Dempe, Annotated bibliography on bilevel programming and mathematical programs with equilibrium constraints.

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

[41] N. Friedman, D. Geiger, M. Goldszmidt, Bayesian network classifiers, Machine learning 29 (2-3) (1997) 131–163.

[42] C. Holmes, N. Adams, A probabilistic nearest neighbour method for statistical pattern recognition, Journal of the Royal Statistical Society: Series B (Statistical Methodology) 64 (2) (2002) 295–306.

[43] J. Pearl, Reverend bayes on inference engines: A distributed hierarchical approach, in: Proceedings of the Second AAAI Conference on Artificial Intelligence, AAAI Press, 1982, pp. 133–136.

[44] J. Pearl, 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, 1985, pp. 15–17.

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

[46] J. R. Quinlan, Decision trees as probabilistic classifiers, in: Proceedings of the Fourth International Workshop on Machine Learning, Elsevier, 1987, pp. 31–37.

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

[48] D. Dubois, H. Prade, Unfair coins and necessity measures: towards a possibilistic interpretation of histograms, Fuzzy sets and systems 10 (1-3) (1985) 15–20.

[49] N. Moha, Y. G. Gueheneuc, L. Duchien, A. F. L. Meur, DECOR: A Method for the Specification and Detection of Code and Design Smells, IEEE Transactions on Software Engineering 36 (1) (2010) 20–36.

[50] N. Tsantalis, A. Chatzigeorgiou, Identification of Move Method Refactoring Opportunities, IEEE Transactions on Software Engineering 35 (3) (2009) 347–367.

[51] R. Gopalan, Automatic detection of code smells in java source code, Ph.D. thesis, University of Western Australia (2012).

[52] C. Y. Ma, X. Z. Wang, Inductive data mining based on genetic programming: Automatic generation of decision trees from data for process historical data analysis, Computers & Chemical Engineering 33 (10) (2009) 1602–1616.

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

[54] S. S. Mullick, S. Datta, S. G. Dhekane, S. Das, Appropriateness of performance indices for imbalanced data classification: An analysis, Pattern Recognition 102 (2020) 107197.

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

[56] K. Deb, R. B. Agrawal, et al., Simulated binary crossover for continuous search space, Complex systems 9 (2) (1995) 115–148.

[57] M. Srinivas, L. M. Patnaik, Genetic algorithms: A survey, computer 27 (6) (1994) 17–26.

[58] E.-G. Talbi, Metaheuristics: from design to implementation, John Wiley & Sons, 2009.

[59] D. Dubois, H. Prade, Possibility theory and data fusion in poorly informed environments, Control Engineering Practice 2 (5) (1994) 811–823.

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

[61] A. Ouni, M. Kessentini, H. Sahraoui, M. Boukadoum, Maintainability defects detection and correction: a multi-objective approach, Automated Software Engineering 20 (1) (2013) 47–79.

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

[63] B. Henderson-Sellers, Object-oriented metrics: measures of complexity, Prentice-Hall, Inc., 1995.

[64] A. E. Eiben, S. K. Smit, Parameter tuning for configuring and analyzing evolutionary algorithms, Swarm and Evolutionary Computation 1 (1) (2011) 19–31.

[65] R. Mallipeddi, P. N. Suganthan, Q.-K. Pan, M. F. Tasgetiren, Differential evolution algorithm with ensemble of parameters and mutation strategies, Applied soft computing 11 (2) (2011) 1–18.

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

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

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

[69] A. Arcuri, L. Briand, A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering, Software Testing, Verification and Reliability 24 (3) (2014) 219–250.

[70] N. Almarimi, A. Ouni, S. Bouktif, M. W. Mkaouer, R. G. Kula, M. A. Saied, Web service api recommendation for automated mashup creation using multi-objective evolutionary search, Applied Soft Computing 85 (2019) 1–13.

[71] J. Cohen, Statistical power analysis for the behavioral sciences, Erlbaum Associates, Hillsdale, 1988.

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

[73] J. Whittle, P. Sawyer, N. Bencomo, B. H. Cheng, J.-M. Bruel, Relax: Incorporating uncertainty into the specification of self-adaptive systems, in: Proceedings of the 17th International Requirements Engineering Conference, IEEE, 2009, pp. 79–88.

[74] K. M. Bowers, E. M. Fredericks, R. H. Hariri, B. H. Cheng, 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 (2020) 1–51.

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

[76] K. Erni, C. Lewerentz, Applying design-metrics to object-oriented frameworks, in: Proceedings of the 3rd international software metrics symposium, IEEE, 1996, pp. 64–74.

[77] R. Marinescu, Detection strategies: Metrics-based rules for detecting design flaws, in: Proceedings of the 20th IEEE International Conference on Software Maintenance, IEEE, 2004, pp. 350–359.

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

[79] R. Marinescu, G. Ganea, I. Verebi, Incode: Continuous quality assessment and improvement, in: Proceedings of the 14th European Conference on Software Maintenance and Reengineering, IEEE, 2010, pp. 274–275.

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

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

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

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

[84] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, H. Sahraoui, A bayesian approach for the detection of code and design smells, in: Proceedings of the 9th International Conference on Quality Software,, IEEE, 2009, pp. 305–314.

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

[86] A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y.-G. Gueheneuc, E. Aimeur, SMURF: A SVM-based incremental anti-pattern detection

approach, in: Proceedings of the 19th Working conference on Reverse engineering,, IEEE, 2012, pp. 466–475.

[87] A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y.-G. Guéhéneuc, G. Antoniol, E. Aïmeur, Support vector machines for anti-pattern detection, in: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering,, IEEE, 2012, pp. 278–281.

[88] S. Hassaine, F. Khomh, Y.-G. Guéhéneuc, S. Hamel, 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, 2010, pp. 343–348.

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

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

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

[92] M. Boussaa, W. Kessentini, M. Kessentini, S. Bechikh, S. B. Chikha, Competitive Coevolutionary Code-Smells Detection, in: Proceedings of the 5th International Symposium on Search Based Software Engineering, Vol. 8084, Springer, 2013, pp. 50–65.

[93] W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, A. Ouni, A Cooperative Parallel Search-Based Software Engineering Approach for Code-Smells Detection, IEEE Transactions on Software Engineering 40 (9) (2014) 841–861.

[94] D. Rapu, S. Ducasse, T. Gîrba, R. Marinescu, Using history information to improve design flaws detection, in: Proceedings of the 8th European Conference on Software Maintenance and Reengineering,, IEEE, 2004, pp. 223–232.

[95] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, A. D. Lucia, D. Poshyvanyk, 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, 2013, pp. 268–278.

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

[97] S. Fu, B. Shen, Code Bad Smell Detection through Evolutionary Data Mining, in: Proceedings of the 2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement,, IEEE, 2015, pp. 1–9.

[98] E. V. Emden, L. Moonen, Java quality assurance by detecting code smells, in: Proceedings of the 9th Working Conference on Reverse Engineering,, IEEE, 2002, pp. 97–106.

[99] G. Langelier, H. Sahraoui, P. Poulin, Visualization-based analysis of quality for large-scale software systems, in: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering,, ACM, 2005, pp. 214–223.

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

[101] E. Hüllermeier, Learning from imprecise and fuzzy observations: Data disambiguation through generalized loss minimization, International Journal of Approximate Reasoning 55 (7) (2014) 1519–1534.

[102] E. Hüllermeier, S. Destercke, I. Couso, Learning from imprecise data: adjustments of optimistic and pessimistic variants, in: International Conference on Scalable Uncertainty Management, Springer, 2019, pp. 266–279.

[103] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, B. Murphy, 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, 2009, pp. 91–100.

[104] K. Li, Z. Xiang, T. Chen, K. C. Tan, 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, 2020, pp. 573–584.

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

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

[107] Z. Zhu, Y. Li, H. Tong, Y. Wang, Cooba: Cross-project bug localization via adversarial transfer learning, in: Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI, 2020, pp. 3565–3571.

[108] M. Fowler, K. Beck, Refactoring: Improving the Design of Existing Code, Addison-Wesely, 1999.

[109] R. C. Martin, Agile software development: principles, patterns, and practices, Prentice Hall, 2002.

[110] R. Wirfs-Brock, A. McKean, Object design: roles, responsibilities, and collaborations, Addison-Wesley Professional, 2003.

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

[112] A. Ouni, A mono-and multi-objective approach for recommending software refactoring, Ph.D. thesis, Faculty of arts and sciences of Montreal (2014).

[113] S. R. Chidamber, C. F. Kemerer, A metrics suite for object oriented design, IEEE Transactions on software engineering 20 (6) (1994) 476–493.

[114] R. Marinescu, Measurement and quality in object oriented design, Ph.D. thesis, Politehnica University of Timisoara (2002).

Appendix A. Description of the processed code smells

During this work, we addressed eight types of anti-patterns stated in Table 9, which are among the most common addressed anti-patterns in the research topics of software maintenance [108], [109], [110], [78], [111], [112], [72]:

Table 9: List of frequently addressed code smells over the detection task into the literature.

Code Smell / Antipattern	**Description**
God Class (aka Blob)	It is found when most of a system's behavior is centralized by a large class, while other classes primarily include data.
Data Class	It appears when a class only stores information without performing any processing.
Feature Envy	It comes up whenever a method accesses another class's data much more than its own.
Long Method	It is found when too many line lines of code are included in one method.
Duplicate code	It arises when a code fragment that looks identical to other code fragments located at many classes.
Long Parameter List	It is found when a method includes a high number of parameters.
Spaghetti Code	This smell is triggered in the case where the code's control structure gets complicated and tangled.
Functional Decomposition	It occurs when a class is built to carry out only one function. This is found in code generated by object-oriented developers who are not experienced.

Appendix B. Description of the adopted metrics

The adopted metrics throughout this work are illustrated in Table 10 [113], [114], [78], [112], [72]:

Table 10: List of the considered measures.

Metric	Description
ANA - Average Number of Ancestors	This metric implies the mean number of classes from which information is inherited by every class.
AOFD - Access Of Foreign Data	This measure is adopted for counting the number of attributes, which are accessed directly from unrelated classes or by calling getters methods.
CAM - Cohesion Among Methods of Class	This measure is employed to compute the relatedness between class methods, calculated by summing the intersection of method parameters with the maximum independent set of all the types of parameters within the class.
CBO - Coupling Between Objects	The number of classes invoking a function or accessing a variable of a particular class is counted by this measure.
CIS - Class Interface Size	Such metric is used for counting the number of methods in a class that is public. It is perceived in a design as the average across all classes.
CM - Changing Method	The number of separate methods that call the method calculated is counted by this metric.
DAM - Data Access Metric	It calculates the ratio of the number of attributes that are private or protected to the overall number of declared attributed throughout the class.
DCC - Direct Class Coupling	The number of different classes is counted by this metric, and to which class is directly related. It includes classes that are directly related to the attribute declarations and passing messages (i.e. parameters) in the methods.
DSC - Design Size in Classes	It is used for counting the overall number of classes existing in the design without taking into account the library classes that are imported.
LOC - Lines of Code	This measure is devoted for measuring a given program size based on counting the instruction number residing within classes or methods.
MFA - Measure of Functional Abstraction	It calculates the ratio of the methods number in which a class inherits to the total methods number that the class's member methods can reach.

Metric	Description
MOA - Measure of Aggregation	The number of data declarations in which their types are user-defined classes is counted by this metric.
NOA - Number of Attributes	This metric is adopted for counting the attributes number belonging to a class existing in the chosen program.
NOAM - Number of Accessor Methods	It is employed for counting the accessor (i.e. getter) and mutator (i.e. setter) numbers that pertain to the class in question.
NOF - Number of Fields	This metric counts the number of fields existing in the classes.
NOH - Number of Hierarchies	It is adopted for counting the entire number of class hierarchies within the design.
NOM - Number of Methods	The number of methods that a class defines is counted by this measure.
NOPA - Number of Public Attributes	This metric is employed for counting the public attributes number belonging to a specific class that exists in a given program.
NPA - Number of Private Attributes	The private attributes number belonging to a given class is measured using this metric.
TCC - Tight Class Cohesion	This measure is adopted for calculating the relative number of method pairs of a class, which have access to at least one of the measured class attributes in common.
WMC - Weighted Methods per Class	It is employed to calculate a class's complexity based on the methods number within the class that exist.
WOC - Weighted Of Class	It counts the functional methods (i.e., non-accessor methods) within a given class divided by the overall interface members number.