A Bi-level Evolutionary Approach for the Multi-label Detection of Smelly Classes

Sofien Boutaib[1], Maha Elarbi[1], Slim Bechikh[1], Fabio Palomba[2], Lamjed Ben Said[1]
[1]SMART Lab, University of Tunis, ISG, Tunisia
[2]Software Engineering (SeSa) Lab, University of Salerno, Italy
mohamedsofien.boutaib@isg.u-tunis.tn

ABSTRACT

This paper presents a new evolutionary method and tool called BMLDS (Bi-level Multi-Label Detection of Smells) that optimizes a population of classifier chains for the multi-label detection of smells. As the chain is sensitive to the labels' (i.e., smell types) order, the chains induction task is framed as a bi-level optimization problem, where the upper-level role is to search for the optimal order of each considered chain while the lower-level one is to generate the chains. This allows taking into consideration the interactions between smells in the multi-label detection process. The statistical analysis of the experimental results reveals the merits of our proposal with respect to several existing works.

CCS CONCEPTS

• **Software and its engineering** → **Search-based software engineering**; **Maintaining software**.

KEYWORDS

Multi-label detection of smells, classifier chains, labels' order, bi-level optimization, evolutionary algorithm.

ACM Reference Format:
Sofien Boutaib[1], Maha Elarbi[1], Slim Bechikh[1], Fabio Palomba[2], Lamjed Ben Said[1]. 2022. A Bi-level Evolutionary Approach for the Multi-label Detection of Smelly Classes. In *Genetic and Evolutionary Computation Conference Companion (GECCO '22 Companion), July 9–13, 2022, Boston, MA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3520304.3528946

1 INTRODUCTION

A code smell corresponds to an anomaly inherent in the source code that shows a violation of fundamental design principles such as hierarchy, abstraction, encapsulation, modifiability, and modularity. Even if the software engineers are aware of the design principles, these latter are frequently broken due to inexperience, time constraint, and intense market competition. The existence of code smells (or Anti-patterns) impedeS the maintenance process of the software and may make bugs more likely over time. For example, some fragments of source code may contain duplicated methods, a large number of lines of code, long parameter lists, etc. If the code smell exists in the source code of software, it means that some fragments of the source code needs to be refactored, which is one of the solution for removing code smells.

According to the literature [1], some approaches can only detect one type of anti-patterns in the source code. Moreover, the code can be classified as either smelly or non-smelly. Nonetheless, a source code fragment could be affected by multiple code smell types. To put it another way, a given code fragment can include more than one anti-pattern such as feature envy and long method. Based on this fact, the code smell detection corresponds to the multi-label classification problem. This latter appears when the instances in a given dataset could be associated with more than one class label simultaneously [2, 3]. The main issues to handle the problem of multi-label classification are the relationships between class labels as well as the imbalance class labels. A little number of approaches have been developed to address the problem of multi-label classification. Moreover, the widespread class imbalance among class labels has not been thoroughly investigated and all of them failed to consider the relationships among labels throughout the building stage. Based on these findings, multi-label code smell detection can be viewed as a bi-level optimization problem in which the label ordering should be optimized and then the (near) optimal detector for multi-label imbalanced data code smell detection should be determined.

In this paper, we propose our approach named BMLDS (Bi-level Multi-Label Detection of Smells). The basic idea of our approach is to evolve the label ordering at the upper level, while optimizing the lower level population that are classifier chains composed of ODTs to deal with the problem of imbalanced class labels problem. We notice that for every upper-level individual that corresponds to a label ordering, the entire lower-level population is evolved to discover the corresponding optimal classifier chains (more precisely code smell detectors).

2 THE PROPOSED APPROACH: BMLDS

In this study, we choose to focus on fifteen code smells/anti-patterns, which are: Duplicate code, Long method, Large class, Dead Code, Feature Envy, Long Parameter List, Spaghetti Code, Functional Decomposition, Bad Comments, Bad naming, Brain method, Refused Parent Bequest, Shotgun Surgery, Speculative Generality, Intensive coupling. These code smells are actually among the used ones in the literature. In addition, we took into account the commonly adopted structural metrics. The first step in constructing our BMLDS approach is to create the Multi-Label Base of Examples (ML-BEs), which is characterized by instances (software classes) that are associated to more than one class label (smell types). To do so, a set of smell detectors (such as DECOR, JDeodorant, etc) are used to construct a base of examples for each considered smell type. Then,

GECCO '22 Companion, July 9–13, 2022, Boston, MA, USA

ACM ISBN 978-1-4503-9268-6/22/07.
https://doi.org/10.1145/3520304.3528946

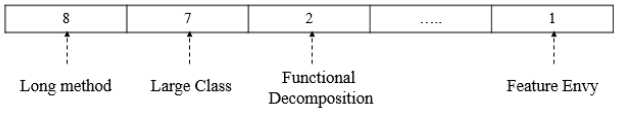

Figure 1: Upper level Solution representation

these bases of examples are merged using the voting fusion to obtain at the end a ML-BEs (Multi-Label Base of Examples). Once the ML-BE is constructed, the objective is to tackle the problem of label ordering which in turn will improve the classification performance. Such a task could be performed based on a hierarchical structure in which the upper level includes the evolution of the label ordering while the lower level contains the optimization of the detectors according to the obtained label ordering as input. We notice that the optimization in both levels is performed based on the GA. After the building process, the obtained detectors are used to identify the existing smell types within the unknown software classes.

2.1 BMLDS building phase

As shown by Figure 2 our approach has an evolutionary schema, it contains two levels: (1) the upper level (optimizes the label ordering) and (2) the lower level (optimizes the chains of the ODTs based on the label ordering obtained from the upper level). The evolution in both levels is performed based on the GA which is composed of four main components that are: (1) Solution representation, (2) variation operators, (3) fitness function, and (4) mating selection. Lets move to present these components on each level.

- **Upper level optimization**
  At this level, the optimization process is performed as follows. First, we choose the solution with the lowest fitness value, and then we modify the values of the fitness that belong to the remaining solutions till reaching the stopping criterion. Then, the mating selection and the variation operators (i.e., crossover and mutation) are applied. We have used ExF-cor (Example-based F-Measure-correlation) as a fitness function to evaluate the solutions. This latter combines the Example-based F-Measure and the correlation ratio to deal with the multi-label problem as well as imbalanced data issue. Moreover, such measure encourages both convergence and diversity at the upper-level, which helps the algorithm to approximate the optimal solutions.

(1) **Solution encoding**: At the upper level, a candidate label ordering solution is encoded as a chromosome in which each gene corresponds to a single label. It is worth noting that every individual is represented by a vector containing 15 integers representing the number of considered class labels existing in the BE and every integer indicates the corresponding label existing in the BE. The value '1' indicates the first class label existing in the BE, while the value '2' the second label in the indicated order within the BE, and so on. To have a better idea, Figure 1 depicts a solution that represents a vector of label ordering.

(2) **Solution evaluation**: The solution evaluation step consists in assessing the classifiers performances, therefore guiding the evolution to individuals that have high performances as they took all class labels (i.e., majority and minority) into consideration at the same time. In the literature, several evaluation measures for the case of multi-label classification problem have been proposed. Among these measures, we can found $ExF - cor$, which is proposed by Moyanoa et al. [5]. Such measure is composed of two measures: ExF (Example-based F-Measure) (cf. Equation 1) and the c_r (coverage ratio measure). The former is similar to the F-measure but has the ability to deal with the multi-label classification problem. The ExF is computed for every instance and after that averaged across all instances. ExF is defined as a number between 0 and 1; the higher the value, the better the performance of the method is. The c_r has been defined in the aim to calculate how many times every label shows up in the ensemble. This measure is depicted in Equation 2, where v is the votes vector, i.e. a vector containing the number of times every label appears in the ensemble, v_w represents a votes vector in the worst scenario, and stdv(v) represents the vector v standard deviation. The worst-case scenario is the one in which the votes vector is as imbalanced as possible, with some labels appearing in the whole classifiers and the rest not appearing at all. To obtain a measure value 0 and 1, the c_r is split by the worst case. In fact, in case where the c_r was not considered in fitness, the labels that are predicted easily would appear frequently within individuals, leading others to appear less frequently. As both measures have values between 0 and 1, the value of the ExF is maximized and the value of the c_r is minimized, the fitness function is presented as linear combination of those measures. The obtained fitness is called $ExF - cor$ (Example-based F-Measure correlation) (cf, Equation 3)

$$\uparrow ExF = \frac{1}{m}\sum_{i=1}^{m}\frac{2 \mid \hat{Y}_i \cap Y_i \mid}{\mid \hat{Y}_i \mid \cup \mid Y_i \mid} \tag{1}$$

$$\downarrow c_r = \frac{stdv(v)}{stdv(v_w)} \tag{2}$$

$$\uparrow ExF - cor = \frac{ExF + (1 - c_r)}{2} \tag{3}$$

(3) **Selection operator**: As BMLDS has a set of label ordering individuals, it should motivate the selection of good individual parents at every generation of the GA while allowing the less effective individuals to be present in the reproduction process in the aim to get away from the local optima and approximate the globally optimum label ordering. Hence, the tournament selection operator is utilized to choose parents for the reproduction. In fact, individuals are chosen as parents at every generation (N/2). This permits for the acceptance of the bad movements within the search space, allowing for the escape from locally optimal label ordering.

(4) **Crossover and mutation operators**: For the variation of the population at the upper level, we adopt the uniform

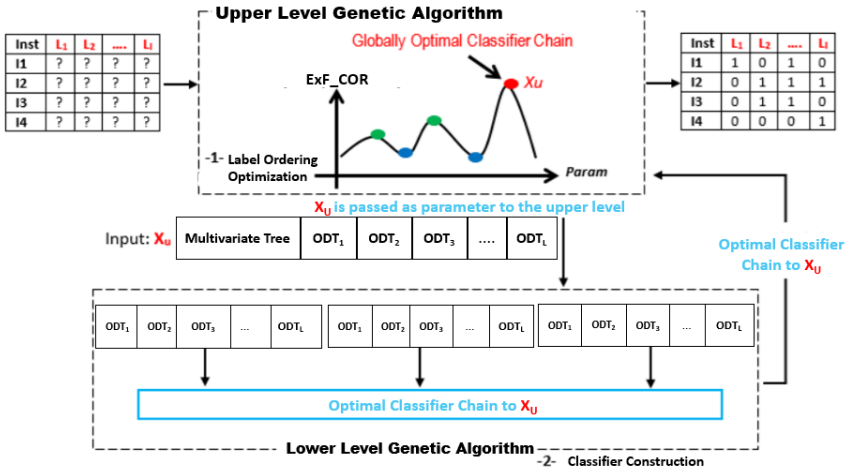

Figure 2: Global schema of the proposed approach

crossover operator since it permits us to vary all pieces of whatever chromosome. We adopt the order crossover, which generate offspring without duplicated class labels. For the mutation, we employ a swap mutation. More precisely, we pick two positions in a random manner a vector of label ordering and swap randomly two class labels within that child.

- **Lower level optimization**
  At this level, we start by the calculation of the normalized values of the objective function. Then, we generate solutions for the passed solution from the upper level. We modify the fitness values, then we eliminate the bad solution and update the fitness values another time. To better, clarify for each passed label ordering from the upper level, a set of classifiers chains are generated and evolved using the variation operators.

(1) **Solution encoding**: In the lower level, each chromosome corresponds to a chain of ODTs (cf. Figure 2). Every ODT is in the chain is destined to a single and specific label. This is performed in the aim to capture the labels dependencies. The classifier chain build 15 ODT classifiers (every one handling one label) connected in a chain, extending the feature space by all the preceding within the chain each time. The choice of the ODT is due to their specificity on the use of oblique splitting hyperplanes and the orthogonal ones to handle the problem of imbalanced data. Figure 2 presents the building of a chain of ODTs. The original features (quality metrics) are used as predictors by the ODT detectors, which is supposed to predict the first class label. The ODT detector 2 that will predict the class label 2, uses the previous label as supplemental input information, catching potential dependencies among the class labels and so on. Every ODT is coded using a two-array format with a breadth-first ordering. The first array holds the weights of the various attributes for a decision (internal node) and the NULL value for a terminal (leaf) node. The second array includes the value of the threshold for every internal node as well as a Boolean cell for every leaf node where '1' represents class 1 and '0' represents class 0. If a node possesses index i in the breadth-first order, then the left child is located at the position 2i and the right child at the position (2i+1). The classification step should then be carried out in a chained manner where the previously generated L ODTs are changed into a sequence of label.

(2) **Solution evaluation**: The evaluation of the lower level solutions is performed based on a fitness function (cf. Equation 4) that is composed of the $ExF - cor$ (cf. Equation 4) and the ODT Depth which is a parameter specified by the software engineer. It is important to note that both measures have the same weights over the proposed fitness function.

$$fitnessfunction = \frac{ExF - cor + Depth}{2} \quad (4)$$

(3) **Solution evaluation**: In the lower-level, we used the tournament selection operator to pick up parents for the reproduction. Half of the population (i.e., N/2) is chosen at every generation.

(4) **Crossover and mutation operators**: For the variation of the population at the lower level, we utilized the order crossover operator that leads to solutions without redundancy. For the mutation, we can have two types of changes at the level of a gene in chromosome (i.e., an ODT): Weigh change and label change. The former modifies the metrics wights by permitting for the addition of a new metric (whose weight changes from zero to non-zero) and the removal of an existing metric (its weight turns into zero), while the latter exchanges two leaf-nodes by switching their pair of labels.

2.2 Application module for optimized detectors

The output of the BMLDS is an optimized detector (i.e., classifier chain). Such detector is ready to be applied in the aim to identify

Table 1: Parameters settings

Parameters	BMLDS	GP	BLOP
Crossover rate	0.9	0.9	0.8
Mutation rate	0.1	0.5	0.5
Population size	200	100	30

the existing smell types . Each software class could be assigned to multiple smell types.

3 EXPERIMENTAL STUDY

Our BMLDS tool is assessed on the basis of a series of commonly adopted open-source Java projects, consisting of Xerces-J, GranttProject, ArgoUML, Ant-Apache, JFreechart, and Azureus. In this paper, we used a five-fold cross-validation strategy to evaluate our approach, which has been compared to the relevant state-of-the-art approaches (DECOR [4], GP [6], BLOP[7]). The results of the generated metrics on the test data are then averaged across the five chunks. In fact, we adopted two performance metrics for classification appropriate for dependent and imbalanced class labels environments to quantify the various methods performances under consideration. The first measure is the fitness function, which was previously discussed in Section 2.1. The second one is the *Macro-G-mean* (Equation 7) measure that corresponds to the geometric mean of *Macro-Average-Specificty* (Equation 5) and *Macro-Average-Sensitivity* (Equation 6). More precisely, it is a metric that assesses the balance of classification performance on the majority classes as well as the minority ones.

$$Macro - Average - Specificity = \frac{1}{q} * \sum_{i=1}^{q} \frac{TN_i}{TN_i + FP_i} \quad (5)$$

$$Macro - Average - Sensitivity = \frac{1}{q} * \sum_{i=1}^{q} \frac{TP_i}{TP_i + FN_i} \quad (6)$$

$$Macro - G - mean = \frac{1}{q} * \sum_{i=1}^{q} \sqrt{Specificty_i * Sensitivity_i} \quad (7)$$

where q is the whole number of instances in the base of examples. The tuning of search algorithms' parameters is an important consideration before running them. Indeed, parameter selection has a significant impact on metaheuristic performance. To address this problem, we utilized the trial-and-error method to fine-tune the BMLDS parameters. The parameters for the other search algorithms were determined using the original works (cf. Table 1). To ensure fairness in comparisons, all search methods used the same stopping criterion. Every run is indeed terminated after 256500 fitness assessments.

Based on Figure 3, BMLDS outperforms all the other considered approaches while the second-best method is BLOP. The three other methods DECOR, GP, and BLOP have shown poor results. These outcomes could be explained by the the fact that the existing approaches do not consider the dependencies between class labels in addition to the co-occurrences of the class labels. Moreover, the data imbalance ratio is high through the identification process and all the existing approaches (including the Baseline Classifier Chain) are inappropriate for this problem. ‘

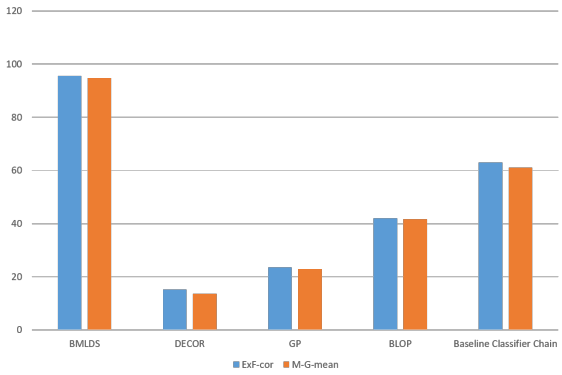

Figure 3: ExF-cor and M-G-mean median scores of BMLDS, DECOR, GP, BLOP, and Baseline Classifier Chain over 31 runs on Large Class smell type identification

4 CONCLUSION

The code smell detection problem is usually considered as a single label classification problem, which is not the real world scenario in which a software class could have more than one smell type. To do so, we have developed BMLDS as as a new evolutionary method and tool that evolves a set of classifier chains for the multi-label detection of smells. Our approach adopts the bilevel schema where the upper-level role being to find the optimal order for every considered chain and the lower-level role being. to produce chains. As future work, we plan to test BMLDS for the identification of smell types that exist on web services as well as the desktop applications.

ACKNOWLEDGEMENT

Dr. Fabio Palomba gratefully acknowledges the support of the Swiss National Science Foundation through the SNF Projects No. PZ00P2_186090

REFERENCES

[1] Muhammad Ilyas Azeem, Fabio Palomba, Lin Shi, and Qing Wang. 2019. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology* 108 (2019), 115–138.
[2] Radhia Azzouz, Slim Bechikh, and Lamjed Ben Said. 2014. A Multiple Reference Point-based evolutionary algorithm for dynamic multi-objective optimization with undetectable changes. In *2014 IEEE Congress on Evolutionary Computation (CEC)*. 3168–3175.
[3] Slim Bechikh, Lamjed Ben Said, and Khaled Ghédira. 2011. Negotiating decision makers' reference points for group preference-based Evolutionary Multi-objective Optimization. In *2011 11th International Conference on Hybrid Intelligent Systems (HIS)*. 377–382.
[4] N. Moha, Y. G. Gueheneuc, L. Duchien, and A. F. Le Meur. 2009. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering* 36, 1 (2009), 20–36.
[5] Jose M Moyano, Eva L Gibaja, Krzysztof J Cios, and Sebastián Ventura. 2019. An evolutionary approach to build ensembles of multi-label classifiers. *Information Fusion* 50 (2019), 168–180.
[6] Ali Ouni, Marouane Kessentini, Houari Sahraoui, and Mounir Boukadoum. 2013. Maintainability defects detection and correction: a multi-objective approach. *Automated Software Engineering* 20, 1 (2013), 47–79.
[7] Dilan Sahin, Marouane Kessentini, Slim Bechikh, and Kalyanmoy Deb. 2014. Code-smell detection as a bilevel problem. *ACM Transactions on Software Engineering and Methodology* 24, 1 (2014), 1–44.