Computing (2024) 106:2585–2612
https://doi.org/10.1007/s00607-024-01294-x

REGULAR PAPER

Multi-label learning for identifying co-occurring class code smells

Mouna Hadj-Kacem[1] · **Nadia Bouassida**[1]

Received: 4 December 2023 / Accepted: 8 May 2024 / Published online: 27 May 2024

Abstract
Code smell identification is crucial in software maintenance. The existing literature mostly focuses on single code smell identification. However, in practice, a software artefact typically exhibits multiple code smells simultaneously where their diffuseness has been assessed, suggesting that 59% of smelly classes are affected by more than one smell. So to meet this complexity found in real-world projects, we propose a multi-label learning-based approach to identify eight code smells at the class-level, i.e. the most sever software artefacts that need to be prioritized in the refactoring process. In our experiments, we have used 12 algorithms from different multi-label learning methods across 30 open-source Java projects, where significant findings have been presented. We have explored co-occurrences between class code smells and examined the impact of correlations on prediction results. Additionally, we assess multi-label learning methods to compare data adaptation versus algorithm adaptation. Our findings highlight the effectiveness of the Ensemble of Classifier Chains and Binary Relevance in achieving high-performance results.

Keywords Multi-label learning · Class code smells · Problem transformation · Algorithm adaptation

Mathematics Subject Classification 68T05 · 68Q32 · 68N01 · 68T10 · 62H30

1 Introduction

Identifying code smells is a fundamental software maintenance activity. Although code smells are characterized as suboptimal design choices [1] that do not affect the functionality of the software, they still pose threats in both short and long terms.

✉ Mouna Hadj-Kacem
mouna.hadjkacem@gmail.com

Nadia Bouassida
nadia.bouassida@isimsf.rnu.tn

[1] Mir@cl Laboratory, Sfax University, Sfax, Tunisia

Numerous studies have delved into the effects of the presence of code smells in software and have highlighted their harmful impact on software quality as a primary consequence [2–4]. Various dimensions of software quality are affected, encompassing maintainability, correctness and program comprehension [5–7]. Additionally, their influence extends to proneness of both faults and changes [8] with a notable association with the emergence of technical debt. A technical debt represents a situation where long-term quality of the code is sacrificed in favour of prioritizing short-term objectives [9]. This in turn results in the accumulation of a growing burden of deferred costs that manifest in the future.

Considering the significant threats posed by code smells, various aspects of their existence within the software have been examined. These aspects include the origins of their introduction [10], their relative severity [11, 12], the identification methods [13–17] and the way to mitigate them through refactoring [18]. Among these aspects, the identification of code smells has gained substantial interest due to its causal relationship with the other mentioned aspects. According to Fontana et al. [15], previous research in code smell identification can be categorized into two primary groups. The first category comprises rule-based approaches which rely on metrics and require domain experts. The second category involves machine learning-based approaches that are based on learning from training data and have an advantage in reducing cognitive load as required in the first category.

In the literature, most of the approaches that fall in the machine learning-based category are designed to identify just one type of code smell within a single software artefact, such as a class or a method. However, in practice, a software artefact typically exhibits multiple code smells simultaneously. In this work [19], the authors have quantified the diffuseness of code smells in projects and found that 59% of smelly classes are affected by more than one smell. As a result, the fact of identifying a single code smell per artefact cannot meet the complexities found in real-world projects. This highlights the importance of addressing the issue from a different perspective and emphasizes the need of adopting approaches that can simultaneously identify multiple code smells within a single artefact.

One of the important approaches for framing this problem is through the utilization of multi-label learning [20]. Multi-label learning is a machine learning method that has the capability to assign multiple labels to a single instance simultaneously. Despite its potential, there has been only a limited number of studies which employed multi-label learning for code smell identification. In the limited number of studies that have used this approach, the focus has typically been on detecting few types of code smells, either at the class or method level. However, the field of code smells is more expansive and encompasses a comprehensive spectrum of distinct types spanning various levels of granularity, i.e. the Fowler's catalogue is composed of 22 types [1]. Therefore, this variety coupled with the high diffuseness of code smells within projects have accentuated the requirement to address both their identification and the co-occurrence among them.

In this paper, we target 8 types of code smells that belong to class-level by means of different multi-label methods: problem transformation, ensemble and algorithm adaptation methods. Specifically, for the problem transformation method, we use Binary Relevance, Classifier Chain, Label PowerSet and HOMER. For ensemble

method, we apply Ensemble of Classifier Chains, Ensemble of Pruned Sets, RAkEL and AdaBoost MH. Lastly, it's worth noting that the algorithm adaptation, unlike the first two methods, has not been employed thus far in the detection of code smells. For this method, we select BPMLL, BRkNN, IBLR-ML and MLkNN. Our experiments were carried out on 30 open-source Java projects that belong to different sizes and domains.

The main contributions of this work are as follows:

- Explore the co-occurrence of code smells at the class-level to show which code smells frequently appear together.
- Investigate the importance of correlations between different code smells and how they influence prediction outcomes.
- Evaluate and compare different multi-label learning methods to determine the most efficient approach for identifying code smell, allowing us to discern whether the results are influenced by the data transformation or the method adaptation.

The reminder of this paper is organized as follows. Section 2 presents the related work of learning-based approaches. Section 3 outlines the methodology applied in this study. Section 4 describes the construction of the dataset, whose type is a multi-label dataset. The experimental findings and discussion are presented in Sect. 5. Section 6 addresses potential threats to the validity of our findings. Finally, in Sect. 7, we conclude the paper.

2 Related work

Based on the number of identified code smells, machine learning-based methods can be classified into two categories. The first category involves the detection of one code smell within an artefact at a time, referred to as single code smell identification (SCSI). The second category involves identifying multiple code smells simultaneously within an artefact, known as multi code smell identification (MCSI). Table 1 lists the studies falling in these categories.

The SCSI category has a larger number of works compared to the MCSI category. Kreimer [21] introduced a decision tree-based method for detecting Long Method and Large Class. The method was evaluated on the IYC system and the WEKA package. Khomh et al. [22, 23] extended the DECOR (DEtection & CORrection) approach [13] to handle uncertainty in the detection process. They transformed rule card specifications into BBNs (Bayesian Belief Networks), introducing BDTEX (Bayesian Detection Expert) based on the GQM (Goal Question Metric) technique. This allowed systematic BBN construction without relying on rule cards. GanttProject and Xerces were used to evaluate the detection of Blob, Functional Decomposition and Spaghetti Code. Hassaine et al. [24] applied artificial immune system algorithms to GanttProject and Xerces for code smell detection by drawing inspiration from the human immune system. Oliveto et al. [25] proposed ABS (Anti-pattern identification using B-Splines), a method using

Table 1 Machine learning-based detection approaches

Works	Category		Algorithms	Code smells
	SCSI	MCSI		
[21]	x		Decision Tree	Class-level: Large Class Method-level: Long Method
[22, 23]	x		Bayesian Belief Networks (BBNs) + Goal Question Metric (GQM)	Class-level: Blob, Functional Decomposition, Spaghetti Code
[24]	x		Artificial Immune System Algorithm	Class-level: Blob, Functional Decomposition, Spaghetti Code
[25]	x		B-Splines	Class-level: Blob
[26, 27]	x		Support Vector Machines (SVM)	Class-level: Blob, Functional Decomposition, Spaghetti Code, Swiss Army Knife
[15]	x		16 Algorithms*	Class-level: God Class, Data Class Method-level: Long Method, Feature Envy
[31]		x	Problem Transformation (Binary Relevance, Classifier Chains) and Ensemble (BaggingML)	Class-level: God Class and Data Class Method-level: Long Method and Feature Envy
[29]	x		Multi-Layered Perceptron	Class-level: God Class Method-level: Feature Envy
[30]		x	Problem Transformation (Binary Relevance, Classifier Chains, Label Combination)	Method-level: Long Method and Feature Envy
[33]		x	HMML: Hybrid Model with Multi-Level code representation	Method-level: Long Method, Long Parameters List, Complex Method, Complex Conditional, Magic Number, Long Identifier, Long Statement, Missing Default, Empty Catch Clause
[28]	x		Naive Bayes, KNN, Multilayer Perceptron, Decision Tree, Logistic Regression, Random Forest	Class-level: God Class, Data Class Method-level: Long Method, Feature Envy

* J48 (with pruned, unpruned and reduced error pruning), JRip, Random Forest, Naïve Bayes, SMO (with Radial Basis Function and Polynomial kernels) and LibSVM (with the two algorithms *C*-SVC and *ν*-SVC in combination with Linear, Polynomial, RBF and Sigmoid kernels)

interpolation curves and metrics to identify anti-pattern signatures for detecting Blob in Java systems. Maiga et al. [26, 27] introduced a support vector machine-based approach and later expanded it by proposing SMURF where practitioner feedbacks are incorporated. Both approaches were validated on ArgoUML, Azureus and Xerces.

Fontana et al. [15] employed 16 machine learning algorithms for detecting four code smells. Notably, the authors found that J48 and Random Forest delivered the highest performance, while support vector machines exhibited lower performance. The experiments were conducted on four datasets extracted from a large repository of 74 software systems. Subsequently, various works widely adopted the four datasets from the latter study. For instance, in this study [28], the authors have utilized the mentioned datasets where they have applied six machine learning models along with two feature selection techniques, which are Chi-squared and Wrapper-based methods. The subject code smells belong to class and method levels.

Barbez et al. [29] have introduced SMAD (SMart Aggregation of Anti-patterns Detectors), a method that unifies three detection tools into a single tool using a boosting ensemble model. The outcomes produced by these tools are aggregated into a single vector, which is then employed as input for a Multi-Layered Perceptron.

Comparing to the SCSI category, the MCSI category has significantly fewer research studies. Guggulothu et al. [30] have explored code smell detection through a multi-label classification approach. They have applied this approach to determine whether an element at the method level exhibited Long Method and Feature Envy. To address this, they have employed three techniques, namely Binary Relevance, Classifier Chains and Label Combination derived from the problem transformation. The authors combined these techniques with different basic classifiers and have adopted two datasets proposed by Fontana et al. [15] for the experiments. Similarly, Kiyak et al. [31] have adopted four datasets put forth by this work [15]. The authors have employed problem transformation and ensemble techniques. In their experiments, these techniques were tested in conjunction with different basic classifiers, including Decision Tree, Random Forest, Naive Bayes, Support Vector Machine and Neural Network. Besides the mentioned machine learning-based approaches, the multi-label problem has been framed as a search-based approach through a bi-level optimization problem in this work [32]. In another work, the problem has been carried out using deep learning, where the authors have introduced a Hybrid Model with Multi-Level code representation (HMML) [33]. This model integrates Graph Convolutional Neural Networks and Bi-directional Long Short-Term Memory Networks with an Attention Mechanism to perform multi-label classification of method-level code smells.

Unlike previous works in the MCSI category that identify a limited number of code smells within a code fragment, our research tackles a more extensive set, specifically eight code smells at the class-level. To accomplish this, we created a large multi-label dataset derived from 30 open-source projects. Concerning the dataset balance, we applied a sampling technique that is tailored for multi-label datasets. Furthermore, we selected various techniques from different multi-label methods, particularly using algorithm adaptation techniques. Our experiments aim to explore not only code smell identification but also the relationships between them. We also

assess the impact of the choice of techniques by inspecting whether the results depend more on data transformation or method adaptation.

3 Methodology

In this section, we will start with formulating our research questions. Based on these inquiries, we will develop our proposed methodology, which will be presented in more detail in the following sub-sections.

Three research questions *RQs* are addressed in this paper:

- *RQ1*: Which code smells frequently co-occur in class-level artefacts?
- *RQ2*: What is the role of correlation in influencing the outcomes of code smell identification?
- *RQ3*: Does the selection of a multi-label learning method significantly impact code smell identification results, with a focus on whether this influence is attributed more to data transformation or method adaptation?

3.1 Overview of the proposed approach

As illustrated in Fig. 1, our process starts with data retrieval from public repositories. The selected projects undergo statistical analysis and subsequent annotation for the identification of specific code smells. This procedure results in dataset creation. However, since our issue is framed as a multi-label learning problem, the dataset takes the form of a multi-label dataset (see Sect. 4). Following that, we conduct quantitative analysis using a variety of multi-label learning techniques to provide answers to the addressed research questions.

3.2 Statistical analysis

Statistical code analysis involves the extraction of software metrics. These metrics are crucial to capture diverse properties of the source code and serve as quantitative measures of different software aspects. Among the wide range of metrics available in the literature, we primarily focus on the Chidamber and Kemerer (CK) metrics [34]. The CK metrics are a set of well-established software measures that provide a thorough analysis of the codebase. We opted for this metric suite based on the findings in the systematic literature review of Azeem et al. [35], where it was identified as the most commonly used metric suite for code smell detection. Including but not limited to, the CK metrics encompass various aspects such as coupling, cohesion and complexity.

In our research, the CK metrics are used as features in our dataset. As a tool to compute these metrics, we employ the CK tool [36], an open-source tool.[1]

[1] https://github.com/mauricioaniche/ck

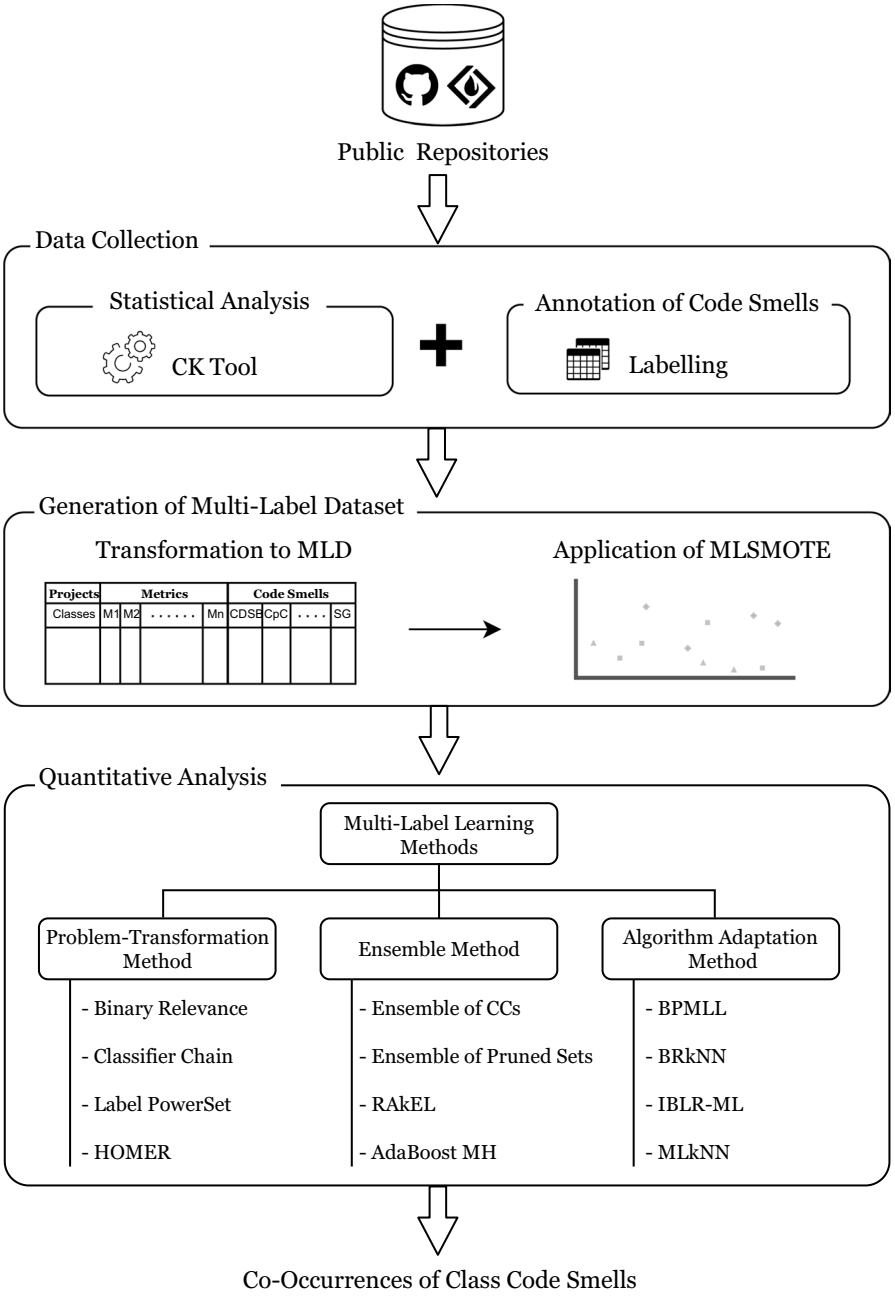

Fig. 1 Overview of the proposed approach

Table 2 Class-level CK metrics

Abbreviation	Metric	Description
abstractMethodsQty	Number of public abstract methods	Counts the number of public abstract methods
anonymousClassesQty	Quantity of anonymous classes	Counts the number of anonymous classes
assignmentsQty	Quantity of assignments	Count the number of assignments
CBO	Coupling between objects	Counts the number of dependencies a class has
cboModified	Coupling between objects	Counts the number of dependencies a class has
comparisonsQty	Quantity of comparisons	Counts number of comparisons
defaultFieldsQty	Number of default fields	Counts the number of public default fields
defaultMethodsQty	Number of public default methods	Counts the number of public default methods
DIT	Depth inheritance tree	Counts the number of fathers a class has
FAN-IN	–	Counts the number of input dependencies a class has
FAN-OUT	–	Counts the number of output dependencies a class has
finalFieldsQty	Number of final fields	Counts the number of public final fields
finalMethodsQty	Number of public final methods	Counts the number of public final methods
innerClassesQty	Quantity of inner classes	Counts the number of inner classes
lambdasQty	Quantity of lambda expressions	Counts the number of lambda expressions
LCC	Loose class cohesion	Calculates the number of indirect connections between visible classes for the cohesion calculation
LCOM	Lack of cohesion of methods	Calculates LCOM metric
LCOM*	Lack of cohesion of methods	Is a normalized metric that computes the lack of cohesion of class within a range of 0 to 1
LOC	Lines of code	Counts the lines of code

Table 2 (continued)

Abbreviation	Metric	Description
logStatementsQty	Number of log statements	Counts the number of log statements in the source code
loopQty	Quantity of loops	Counts the number of loops
mathOperationsQty	Quantity of math operations	Counts the number of math operations
maxNestedBlocksQty	Max nested blocks	Calculates the highest number of blocks nested together
modifiers	Modifiers	Counts the modifiers of class
NOC	Number of children	Counts the number of immediate subclasses that a particular class has
NOF	Number of fields	Counts the number of fields
NOM	Number of methods	Counts the number of methods
NOSI	Number of static invocations	Counts the number of invocations to static methods
numbersQty	Quantity of number	Calculates the number of numbers
parenthesizedExpsQty	Quantity of parenthesized expressions	Counts th umber of expressions inside parenthesis
privateFieldsQty	Number of private field	Counts the number of private fields
privateMethodsQty	Number of private methods	Counts the number of private methods
protectedFieldsQty	Number of protected fields	Counts the number of public protected fields
protectedMethodsQty	Number of public protected methods	Counts the number of public protected methods
publicFieldsQty	Number of public fields	Counts the number of public fields
publicMethodsQty	Number of public methods	Counts the number of public methods
returnQty	Quantity of returns	Counts the number of return instructions
RFC	Response for a class	Counts the number of unique method invocations in a class
staticFieldsQty	Number of static fields	Counts the number of static fields
staticMethodsQty	Number of static methods	Counts the number of static methods
stringLiteralsQty	String literals	Counts the number of string literals
synchronizedFieldsQty	Number of synchronized fields	Counts the number of public synchronized fields
synchronizedMethodsQty	Number of public synchronized methods	Counts the number of public synchronized methods
TCC	Tight class cohesion	Measures the cohesion of a class with a value range from 0 to 1

Table 2 (continued)

Abbreviation	Metric	Description
tryCatchQty	Quantity of try/catches	Counts the number of try/catches
uniqueWordsQty	Number of unique words	Counts the number of unique words in the source code
variablesQty	Quantity of Variables	Counts the number of declared variables
visibleMethodsQty	Number of public visible methods	Counts the number of public visible methods
WMC	Weight method class	Counts the number of branch instructions in a class

Specifically, this tool analyses Java projects through a static analysis where it focuses on the source code rather than the compiled version. It encompasses both class-level and method-level code metrics, but our focus on class code smells leads us to employ only class-level metrics, as outlined in Table 2.

3.3 Annotation of code smells

The annotation of code smells is accomplished through the labelling of true positive samples. The process of labelling, also called oracle creation, is a challenging and time-consuming task demanding significant expertise in this specific field. The resultant oracle serves as the basis for assessing the performance of learning models. There exist three different approaches to ensure the labelling task [37]: (i) the manual approach involves experienced developers who should have considerable knowledge in analysing software design problems, (ii) the tool-based approach is based on existing code smell detection tools, and (iii) the mixed approach which combines both first approaches where the detection results of tools are subsequently evaluated by developers.

More recently, a systematic literature review has been conducted on code smells datasets [38]. The authors have compared between the existing datasets according to different factors including availability, data source, recency and completeness of labelling. Considering these criteria, the authors have selected two datasets as the most comprehensive and adequate code smells datasets:

- The first dataset, proposed by Palomba et al. [3], involves the labelling of 13 types of code smells at both class and method levels across different releases of 30 open source projects. The labelling approach is mixed, wherein a tool is employed to detect code smells in order to generate a list of potential candidates. Subsequently, these results undergo manual validation to classify them as true or false positives.
- The second dataset, called MLCQ, is proposed by Madeyski et al. [39]. This dataset focuses on four code smells, comprising two at the class-level and two at the method level. Instances of code smells are extracted from 792 industry-relevant projects. The oracle for this dataset is manually curated by experienced developers with industrial expertise.

Between these two datasets, although they are both large and heterogeneous, the first dataset stands out by encompassing a greater diversity of code smells. In our study, the problem is casted as a multi-label learning approach, aiming to simultaneously identify multiple code smells. To address this, a substantial number of labels, i.e., distinct code smell types are required. So, the first dataset [3] is particularly advantageous in this context as it effectively provides a more extensive range of identified code smells in its oracle, aligning well with our research objectives.

In our study, the selected dataset comprises 13 code smells at both class and method levels. We have selected 8 code smells that pertain to individual

Table 3 Description of subject class code smells

Code smells	Definition
Class Data Should Be Private	A class that exposes its attributes, thereby violating the principle of data hiding
Complex Class	A class that has a high cyclomatic complexity where it does more than it should
Large Class	Also known as God Class, it dominates most of the system behaviour by implementing numerous responsibilities
Lazy Class	A class that implements minimal functionality and contains few methods
Message Chain	A sequence of invocations that indicate a high level of coupling
Middle Man	A class that delegates most of its implemented functionalities to other classes
Spaghetti Code	A class that implements complex methods which interact between them
Speculative Generality	A class declared as abstract that is unused in the source code

fragments and excluded the smells at method-level and the smells that consider involved classes for their introduction. Further details of the selected smells can be found in Table 3.

3.4 Multi-label learning methods

Learning from multi-label data can be achieved using various methods, including problem transformation, ensemble and algorithm adaptation methods [40]. Each method involves distinct techniques for its application. In our study, we have used all three methods and for each, we have selected four different techniques.

- *Problem Transformation Method (PTM):*
  In the problem transformation method, a multi-label problem is converted into one or more single-label classification tasks [20, 41]. This transformation allows the application of traditional supervised machine learning classifiers which are originally designed for single-label problems. The outputs of these single-label classifiers are then aggregated to address the objective of the initial multi-label classification problem. For this method, we have selected four different techniques:
  - Binary Relevance (BR) [20]: The multi-label dataset is decomposed into multiple binary datasets, where each corresponds to one label. Next, a single-label learning algorithm is employed to address each individual binary dataset.
  - Classifier Chain (CC) [40]: It operates by connecting binary classifiers in a chain in order to address label correlation. Each binary classifier includes the previous predicted labels as supplementary information.
  - Label PowerSet (LP) [42]: Also known as Label Combination, it transforms a multi-label dataset into a multi-class dataset. It creates a new class

for each distinct combination of labels, treating each combination as a unique class in a multi-class problem.
- Hierarchy Of Multi-label classifiERs (HOMER) [43]: It constructs a hierarchy of classifiers with various label combinations while demonstrating its prediction performance.

The primary distinction among these four problem transformation techniques lies in their conservation of label correlation. Binary Relevance stands out as the only technique that does not preserve label correlation.

- *Ensemble Method (EM):*
  The ensemble method involves the combination of several classifiers [44]. The selected techniques are:
  - Ensemble of Classifier Chains (ECC) [40]
  - Ensemble of Pruned Sets (EPS) [45]
  - RAndom k-labELsets (RAkEL) [44]
  - AdaBoost MH [46]
- *Algorithm Adaptation Method (AAM):*
  This method adapts existing single-label classification algorithms to directly handle multiple labels [20, 41]. Unlike the problem transformation method, the algorithm adaptation is classifier dependent. The selected algorithms belong to different learning families:
  - Back-Propagation Multi-Label Learning (BPMLL) [47]
  - BRkNN [48]: Binary Relevance implementation of the k Nearest Neighbours algorithm
  - Instance-Based Learning by Logistic Regression-ML (IBLR-ML) [49]
  - Multi-Label k Nearest Neighbours (MLkNN) [50]

4 Dataset construction

In this section, we describe the construction of the multi-label dataset, encompassing project details and the dataset generation process. Subsequently, we extract the dataset characteristics given their importance in the experiments.

4.1 Generation of multi-label dataset

As mentioned in Sect. 3.3, we will use 30 open-source Java projects from the selected dataset. These projects are heterogeneous as they vary in size and belong to diverse application domains. They can be downloaded from GitHub[2] and SourceForge.[3] The complete list of these projects is provided in Table 4.

[2] https://github.com
[3] https://sourceforge.net/

Table 4 Description of 30 open-source projects

Projects	Release	#Classes	KLOC	Description
Apache Ant	1.8.3	855	128	Java based build tool
Apache Cassandra	1.1	624	110	Database management system
Apache Derby	10.9	1989	444	Relational database management system
Apache Hadoop	0.9	277	50	A framework for distributed datasets
Apache HBase	0.94	732	269	Distributed database system
Apache Hive	0.9	1193	201	Data warehouse software
Apache Ivy	2.1.0	369	51	A tool for managing project dependencies
Apache Karaf	2.3	492	43	Runtime container for applications deployment
Apache Lucene	3.6	2388	375	Search engine software library
Apache Nutch	1.4	250	35	Web crawler
Apache Pig	0.8	945	184	Platform for analysing large datasets
Apache Qpid	0.18	1606	194	Messaging tool
Apache Struts	2.3.4	1274	143	MVC framework
Apache Wicket	1.4.20	1135	170	Java web application framework
Apache Xerces	2.1.0	776	121	XML parser
ArgoUml	0.34	1769	174	UML diagram generator
aTunes	2.0.0	648	55	Player and audio manager
Eclipse Core	3.6.0	1194	270	Integrated development environment
Elasticsearch	0.19	2167	206	RESTful Search and Analytics Engine
Freemind	0.9.0	465	53	Mind-mapping software
Hibernate	4.1.8	236	35	Java persistence framework
HsqlDB	2.2.8	462	151	HyperSQL database engine
Incubating	0.6	535	87	Codebase
Jboss	6.0.0	2434	373	Application server
jEdit	4.5	567	96	Programmer's text editor
JFreeChart	1.0.14	617	124	Java chart library
JHotDraw	7.6	613	77	Graphic framework
JSL	0.99n	10	0.5	Java Service Launcher
jVLT	1.3.2	272	23	Vocabulary Learning Tool
Sax	2.0	38	0.3	XML Parser

As illustrated in Fig. 2, the process begins with each project undergoing a statistical analysis using the CK tool. Classes are extracted as dataset instances and CK metrics are computed for each one of them. Following that, the labelling is conducted based on the adopted oracle where each type of the eight code smells corresponds to a label. After processing all projects and code smells, at this stage, binary datasets are created where each instance has a single label. These binary datasets are then merged according to the code smell type across all projects. Ultimately, these binary datasets are transformed into one unified multi-label dataset (MLD), where each instance is associated with 8 labels representing code smells.

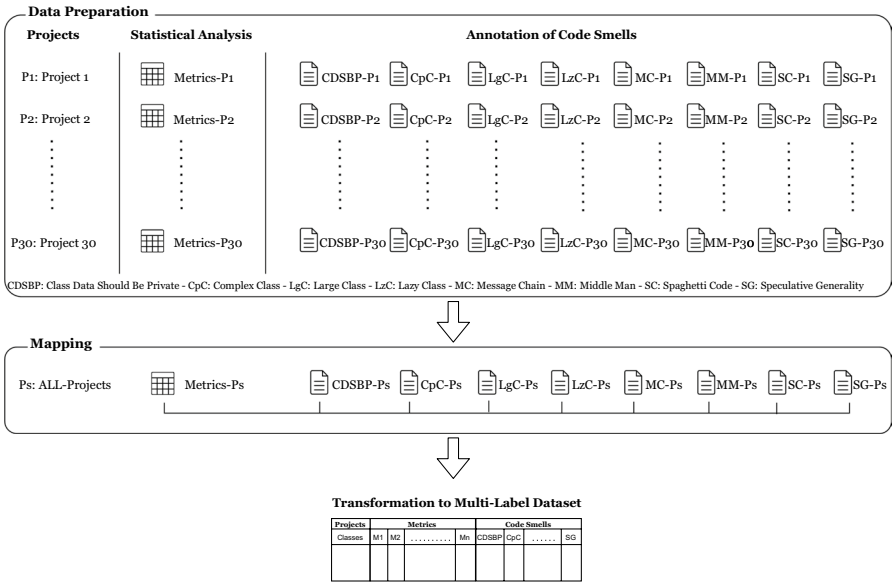

Fig. 2 Construction of multi-label dataset

4.2 Characteristics of multi-label dataset

There exist various metrics that capture specific characteristics of a multi-label dataset. These metrics provide important information, such as label distribution, inter-label relationship and imbalance level [20, 51]. These information serve as crucial criteria for subsequent experimental steps. The most important metrics include:

- *Cardinality* assesses the average number of active labels Y_i per sample, with D is the dataset and N representing the number of instances.

$$Card(D) = \frac{1}{N}\sum_{i=1}^{N}|Y_i| \tag{1}$$

- *Density* is the *cardinality* divided by the number of labels $|\mathcal{L}|$.

$$Dens(D) = \frac{Card(D)}{|\mathcal{L}|} = \frac{1}{|\mathcal{L}|}\frac{1}{N}\sum_{i=1}^{N}|Y_i| \tag{2}$$

- *Mean Imbalance Ratio* (*MeanIR*) describes the mean ratio of imbalance among the labels $\mathcal{L}$. The higher the value is, the more imbalanced the MLD is.

Table 5 Characteristics of MLD

Characteristics	Values
# Samples	26,932
# Labels	8
# Label sets	23
Cardinality	0.048
Density	0.006
MeanIR	13.357

$$MeanIR = \frac{1}{|\mathcal{L}|} \sum_{l \in \mathcal{L}} IRLbl(l) \quad where \quad IRLbl(l) = \frac{\max_{l' \in \mathcal{L}} (\sum_{i=1}^{N} [\![l' \in Y_i]\!])}{\sum_{i=1}^{N} [\![l \in Y_i]\!]} \tag{3}$$

To calculate these metrics, we have used mldr package [52]. As shown in Table 5, the MLD comprises 23 distinct label sets, representing possible combinations. According to Charte et al. [51], the MLD is considered as imbalanced if its MeanIR surpasses 1.5. Following this, our MLD is imbalanced, which means that some labels have high frequency while others are less represented. To deal with, we have applied Multilabel Synthetic Minority Over-sampling Technique (MLSMOTE) [53]. MLSMOTE is a multi-label oversampling algorithm able to generate synthetic instances based on a randomly chosen instances that include minority labels and their nearest neighbour instances.

5 Experiments and results

In this section, we will present the results and discuss the research questions that have been addressed. But before doing so, we will provide the context of the experimentation by presenting the experimental set-up and evaluation metrics.

5.1 Experimental settings

In our experimentation, we have applied three multi-label learning methods: PTM, EM and AAM. Within PTM, we selected BR, CC, LP and HOMER. For EM, we employed ECC, EPS, RAkEL and AdaBoost MH. And for the AAM, we choose BPMLL, BRkNN, IBLR-ML and MLkNN. Some of these techniques necessitate a basic classifier to be implemented, for which we opted for the Random Forest classifier. Our selection is motivated by two key factors. Firstly, Random Forest has yielded significant results in this study [30], particularly in effectively detecting both Long Method and Feature Envy. Secondly, our focus was directed towards exploring diverse multi-label techniques built upon the same basic classifier, aiming for a comprehensive evaluation of these techniques.

The implementation of these techniques was accomplished using MULAN 1.5.0 [54], a Java library designed for learning from multi-label data. MULAN is built on the WEKA library [55] and provides a diverse range of classification and ranking algorithms. Concerning the validation, we utilized 5-fold cross-validation approach where the dataset is split into five folds, i.e. four folds used for training and the remaining fold for testing.

5.2 Evaluation metrics

The assessment of multi-label learning techniques involves distinct metrics compared to single-label learning. Due to the association of each sample with multiple labels simultaneously, evaluating performance in multi-label learning is more complex where the metrics fall into two broad categories: example and label-based metrics [56]. Example-based metrics involve averaging differences between actual and predicted label sets across the samples in the dataset. This category includes two sub-categories: classification metrics (*SubsetAccuracy*, *HammingLoss*, $F-Measure$, *Accuracy*) and ranking metrics (*Coverage*, *AveragePrecision*, *RankingLoss*). In the second category, label-based metrics, the performance for each label is calculated individually and then averaged over all labels (*Macro/MicroAveraging*) [20, 57]. In the equations, for a given instance (x_i), (Z_i) denotes the set of predicted labels and (Y_i) denotes the set of actual labels. The total number of instances and the total number of labels are respectively represented by (N) and ($|\mathcal{L}|$).

- Example-based classification metrics
  - *HammingLoss* ($\searrow$) is the symmetric difference (Δ) between predicted (Z_i) and actual labels (Y_i). It is averaged over total number of labels ($|\mathcal{L}|$) and total number of instances (N). Lower *HammingLoss* indicates better performance.

$$HammingLoss = \frac{1}{N}\frac{1}{|\mathcal{L}|}\sum_{i=1}^{N}\left|Y_i \Delta Z_i\right| \tag{4}$$

  - *SubsetAccuracy* ($\nearrow$), also called *ExactMatchRatio*, known as one of the most strict evaluation measurements. It evaluates the proportion of accurately classified samples accross all the samples, where the predicted label set matches the actual labels.

$$SubsetAccuracy = \frac{1}{N}\sum_{i=1}^{N}[\![Y_i = Z_i]\!] \tag{5}$$

  - $F-Measure$ ($\nearrow$) represents the harmonic mean of *precision* and *recall*. *Precision* is the ratio of correctly predicted labels to the total number of actual labels, averaged across all instances, while *recall* is the ratio of correctly predicted labels to the total number of predicted labels, averaged across all instances.

$$F - measure = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{6}$$

- *Accuracy* (↗) is the proportion of correctly predicted labels to the total number of labels for an instance.

$$Accuracy = \frac{1}{N}\sum_{i=1}^{N}\frac{\left|Y_i \cap Z_i\right|}{\left|Y_i \cup Z_i\right|} \tag{7}$$

- Example-based ranking metrics
  - *Coverage* (↘) is a metric that measures, on average, how a learning algorithm needs to go in the ranked list of predictions to cover all the true labels of an instance. A lower coverage value indicates better performance.

$$Coverage = \frac{1}{N}\sum_{N}^{i=1} \underset{y \in Y_i}{argmax}(rank(x_i, y)) - 1 \tag{8}$$

  - *AveragePrecision* (↗) calculates the proportion of relevant labels ranked before each label and then makes the average across all relevant labels.

$$AveragePrecision = \frac{1}{N}\sum_{N}^{i=1}\frac{1}{\left|Y_i\right|}\sum_{y \in Y_i}\frac{\left|\left\{y' | rank(x_i, y') \leqslant rank(x_i, y), y' \in Y_i\right\}\right|}{rank(x_i, y)} \tag{9}$$

  - *RankingLoss* (↘) measures the proportion of label pairs that are incorrectly ordered in reverse.

$$RLoss = \frac{1}{N}\sum_{N}^{i=1}\frac{1}{\left|Y_i\right|\left|\overline{Y_i}\right|}\left|y_a, y_b : rank(x_i, y_b) > rank(x_i, y_b), (y_{a,b}) \in Y_i \times \overline{Y_i}\right| \tag{10}$$

- Label-based metrics
  - *Micro/Macro averaging* (↗), the macro approach computes the metric for each label and averages the values over all labels, while the micro approach considers predictions for all instances together by aggregating *TP*, *TN*, *FP*, *FN* values for all labels and then calculates the measure across all labels.

$$F1 - Macro = \frac{1}{|\mathcal{L}|}\sum_{l \in \mathcal{L}} F1(TP_l, FP_l, TN_l, FN_l) \tag{11}$$

$$F1 - Micro = F1\left(\sum_{l \in \mathcal{L}} TP_l, \sum_{l \in \mathcal{L}} FP_l, \sum_{l \in \mathcal{L}} TN_l, \sum_{l \in \mathcal{L}} FN_l\right) \tag{12}$$

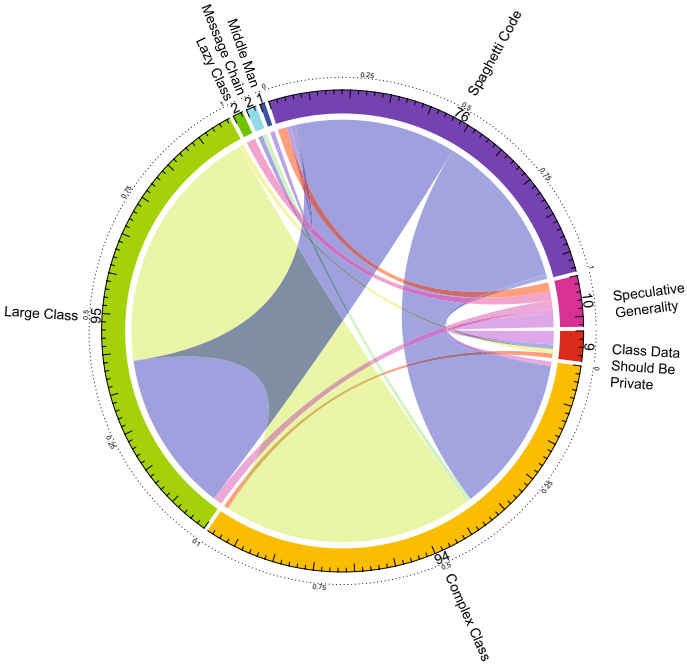

Fig. 3 Chord diagram of class code smells co-occurrences

5.3 Co-occurrence of code smells at class-level

To address the first research question *RQ1*, we opted for the utilization of a chord diagram to offer a comprehensive understanding of potential co-occurrences among two or more code smells at the class-level. Chord diagrams serve as a visual representation to connect entities through chords, providing a graphical representation of relationships. It is important to note that the thickness of these chords reflects the frequency of co-occurrences between code smells: thicker chords denote more frequent co-occurrences, while thinner lines suggest less common associations.

In Fig. 3, the chord diagram presents a graphical depiction of 8 entities, i.e. class code smells, interconnected by chords to illustrate the strength of their relationships. Notably, our analysis reveals that, on average, half of the instances affected by Spaghetti Code exhibit co-occurrences with Large Class, while the remaining half co-occurs with Complex Class. We also found that, approximately, 35% of smelly instances of Complex Class tend to co-occur with Spaghetti Code, while others demonstrate associations with Message Chain, with the remaining linked to Large Class. Through our analysis, three prominent and recurrent co-occurrences have emerged: {Complex Class and Large Class}, {Spaghetti Code, Large Class and Complex Class} and {Spaghetti Code and Large Class}.

The presence of the pair {Complex Class and Large Class} indicates a consistent relationship, implying that when Complex Class is present in a class, the likelihood of Large Class being present is notably high. The frequent co-occurrence

Table 6 Example-based classification results

	Subset Accuracy	Hamming Loss	F-Measure	Accuracy
HOMER	0.9573±0.0014	0.0060±0.0002	0.9592±0.0015	0.9587±0.0015
BR	0.9624±0.0022	0.0051±0.0003	0.9637±0.0020	0.9634±0.0020
LP	0.9618±0.0027	0.0053±0.0004	0.9629±0.0025	0.9626±0.0025
CC	0.9612±0.0017	0.0053±0.0003	0.9630±0.0017	0.9626±0.0017
RAkEL	0.9584±0.0035	0.0058±0.0005	0.9605±0.0034	0.9600±0.0034
ECC	0.9627±0.0018	0.0052±0.0002	0.9641±0.0016	0.9638±0.0016
EPS	0.9618±0.0018	0.0053±0.0003	0.9635±0.0019	0.9631±0.0019
AdaBoost MH	0.9558±0.0013	0.0061±0.0002	0.9558±0.0013	0.9558±0.0013
BPMLL	0.9152±0.0197	0.0118±0.0026	0.9170±0.0196	0.9166±0.0196
BRkNN	0.9575±0.0022	0.0058±0.0003	0.9583±0.0022	0.9581±0.0022
IBLR-ML	0.9566±0.0020	0.0059±0.0003	0.9580±0.0019	0.9577±0.0019
MLkNN	0.9572±0.0020	0.0058±0.0003	0.9582±0.0019	0.9579±0.0020

Table 7 Example-based ranking results

	Coverage	Average Precision	Ranking Loss
HOMER	0.0787±0.0039	0.7162±0.0242	0.0101±0.0007
BR	0.0355±0.0040	0.8301±0.0099	0.0043±0.0005
LP	0.1004±0.0100	0.6168±0.0112	0.0135±0.0014
CC	0.0358±0.0045	0.8322±0.0136	0.0042±0.0006
RAkEL	0.0743±0.0086	0.7018±0.0181	0.0094±0.0011
ECC	0.0173±0.0032	0.9118±0.0092	0.0018±0.0004
EPS	0.0411±0.0046	0.7950±0.0187	0.0050±0.0005
AdaBoost MH	0.1050±0.0078	0.5090±0.0079	0.0143±0.0011
BPMLL	0.0309±0.0042	0.8003±0.0453	0.0037±0.0006
BRkNN	0.0607±0.0084	0.7544±0.0173	0.0078±0.0011
IBLR-ML	0.0388±0.0048	0.7868±0.0119	0.0048±0.0006
MLkNN	0.0391±0.0048	0.7920±0.0147	0.0048±0.0006

of these two smells often stems from their inherent software design characteristics. For instance, a Complex Class, characterized by its high cyclomatic complexity, may tend to assume multiple responsibilities, thereby giving rise to the appearance of the Large Class smell. The additional occurrences of {Spaghetti Code, Large Class and Complex Class} and {Spaghetti Code and Large Class} highlight important patterns for developers to consider during maintenance activities. This may suggest that when a class is affected by the first co-occurrence of Large Class and Complex Class, it tends to lead to the emergence of lengthy methods, thereby elevating the overall complexity of the class. This, in turn, may ultimately pave the way for the emergence of Spaghetti Code. Instances where an artefact is affected by more than one code smell are considered critical, emphasizing the need for a high priority in the refactoring process.

Table 8 Label-based results

	Micro-avg P	Micro-avg R	Micro-avg F1	Macro-avg P	Macro-avg R	Macro-avg F1
HOMER	0.5087±0.0160	0.3872±0.0372	0.4391±0.0289	0.3422±0.0738	0.3067±0.0663	0.3201±0.0698
BR	0.6761±0.0431	0.3113±0.0410	0.4246±0.0406	0.4660±0.0947	0.2700±0.0748	0.3165±0.0802
LP	0.6589±0.0625	0.2750±0.0287	0.3866±0.0326	0.4270±0.0829	0.2403±0.0674	0.2867±0.0733
CC	0.6312±0.0337	0.3197±0.0317	0.4231±0.0271	0.4266±0.0698	0.2786±0.0730	0.3210±0.0731
RAkEL	0.5375±0.0427	0.3958±0.0317	0.4554±0.0328	0.3619±0.0869	0.3138±0.0902	0.3297±0.0870
ECC	0.6354±0.0289	0.3642±0.0339	0.4622±0.0312	0.3959±0.0683	0.2862±0.0623	0.3155±0.0588
EPS	0.6153±0.0386	0.3746±0.0292	0.4641±0.0204	0.4040±0.0938	0.2946±0.0749	0.3171±0.0681
AdaBoost MH	0.0000±0.0000	0.0000±0.0000	0.0000±0.0000	0.0500±0.0612	0.0500±0.0612	0.0500±0.0612
BPMLL	0.2313±0.0843	0.3117±0.1104	0.2380±0.0472	0.2069±0.0503	0.2477±0.0424	0.1936±0.0527
BRkNN	0.7042±0.1039	0.0945±0.0229	0.1663±0.0378	0.4286±0.1898	0.1240±0.0739	0.1658±0.0799
IBLR-ML	0.5843±0.0586	0.1438±0.0134	0.2303±0.0184	0.3922±0.0902	0.1683±0.0686	0.2184±0.0715
MLkNN	0.6715±0.0795	0.1058±0.0246	0.1814±0.0355	0.3609±0.1218	0.1389±0.0745	0.1809±0.0787

5.4 Role of correlation between code smells in prediction results

The presence or absence of correlations between specific code smells can significantly impact the code smell identification task. To delve into this aspect, multi-label learning algorithms may either preserve or ignore these correlations during the detection process. To address this, in the context of *RQ2*, our investigation involves a comparative analysis between algorithms that preserve correlations and those that do not based on various evaluation metrics.

Tables 6, 7 and 8 present the identification results categorized by different evaluation metrics. Table 6 focuses on example-based classification metrics. Among these metrics, Subset Accuracy represents the strictest metric, where ECC is the top-performing algorithm, closely followed by BR, with more than 0.96. Similarly for Accuracy and F-measure metrics, ECC and BR demonstrate superior performance. For Hamming Loss, where lower values approaching zero are desired for optimal results, BR outperforms ECC.

Table 7 presents the evaluation of algorithms using three example-based ranking metrics. Lower values for Coverage and Ranking Loss indicate better performance. Notably, the ensemble technique ECC provides the top performance, followed by the adaptation algorithm technique BPMLL. For the Average Precision, the problem transformation technique CC and its ensemble version exhibit superior performance followed by BR. Across all the three evaluation metrics, AdaBoost MH is ranked as the lowest-performing algorithm.

In Table 8, in contrast to the results in Tables 6 and 7, the results show a decrease for both macro and micro averaging metrics. This shift occurs because label-based metrics are computed for each label rather than each instance. Regarding micro-averaged F-measure, the top three algorithms are associated with the ensemble method: EPS, followed by ECC and RAkEL. On the other hand, for macro-averaged F-measure, RAkEL takes the first place, followed by CC and HOMER. Across all metrics in both macro and micro averaging, AdaBoost MH consistently ranks as the lowest-performing algorithm.

To establish a comprehensive comparison, we employed both the Friedman test and the post-hoc Nemenyi test [58] to evaluate the overall performance of all algorithms across various metric categories. The test aids in determining significant differences between algorithms based on their rankings, enabling the identification of statistically significant variances among pairs of algorithms. As depicted in Fig. 4, the average ranking of each technique within every classification metric category is utilized to determine which classifiers exhibit superior performance compared to others. The statistical significance test highlights that ECC, BR and CC are considered as the top-ranked algorithms, while AdaBoost MH is consistently ranked as the lowest-performing algorithm across all evaluation categories. In summary, addressing *RQ2* regarding the potential impact of the considered correlation on algorithm performance, our analysis suggest that the correlation taken into account by the algorithms does not have a substantial effect on their overall performance.

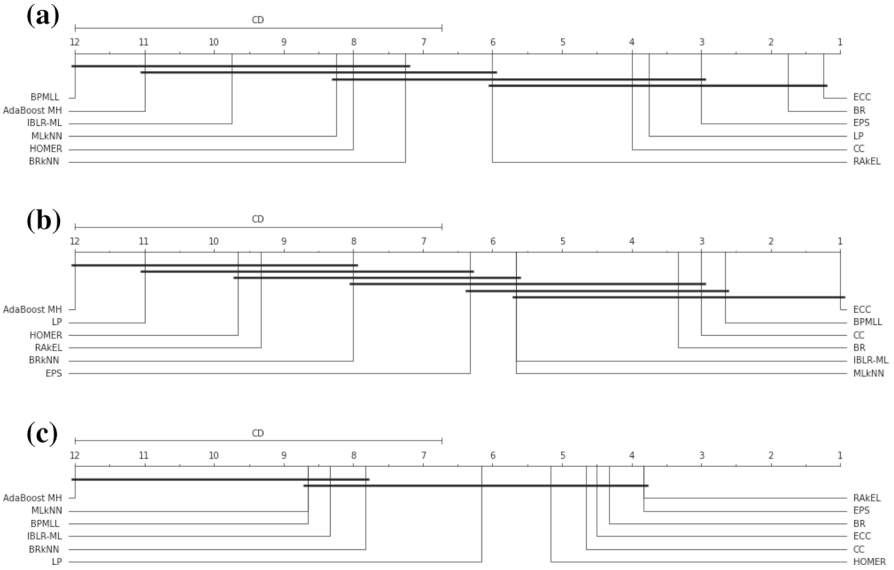

Fig. 4 Average rank diagrams for (**a**) example-based classification metrics, (**b**) example-based ranking metrics and (**c**) label-based metrics

5.5 Comparing multi-label learning methods

In the last question, *RQ3* focus into another factor that potentially influences the results by taking a higher level of abstraction compared to the previous question. Instead of focusing on individual algorithms, *RQ3* centers on the broader method categories to which these algorithms belong. As mentioned earlier, the algorithms fall into three categories: PTM, EM and AAM. In the first two methods, PTM and EM, the multi-label dataset undergoes a transformation to suit problem-solving, while in AAM, the algorithm is adapted to directly operate on the multi-label dataset. To conduct a comprehensive comparison, boxplots are employed across all evaluation metrics, as illustrated in Fig. 5.

The boxplot diagrams concerning example-based classification and label-based metrics showed close distributions, with PTM slightly outperforming EM in terms of variance reduction, followed by AAM. In terms of subset accuracy, F-measure and accuracy, the mean value achieved by PTM is 0.96, whereas AAM attains 0.94. Regarding Hamming loss, where lower values indicate better performance, there is a notable distinction, PTM yields a mean value of 0.005425, whereas AAM stands at 0.007325. However, for example-based ranking metrics, PTM and EM exhibited the largest interquartile range compared to AAM. Specifically, in the Coverage measure, the values of EM range between 0.0173 and 0.105, while for AAM, it falls between 0.0309 and 0.0607. We found that the same observations in Coverage apply to Ranking Loss and Average Precision, where lower values denote superior performance. Notably, techniques within the AAM category demonstrated a closer range by reflecting less variability compared to the other two methods.

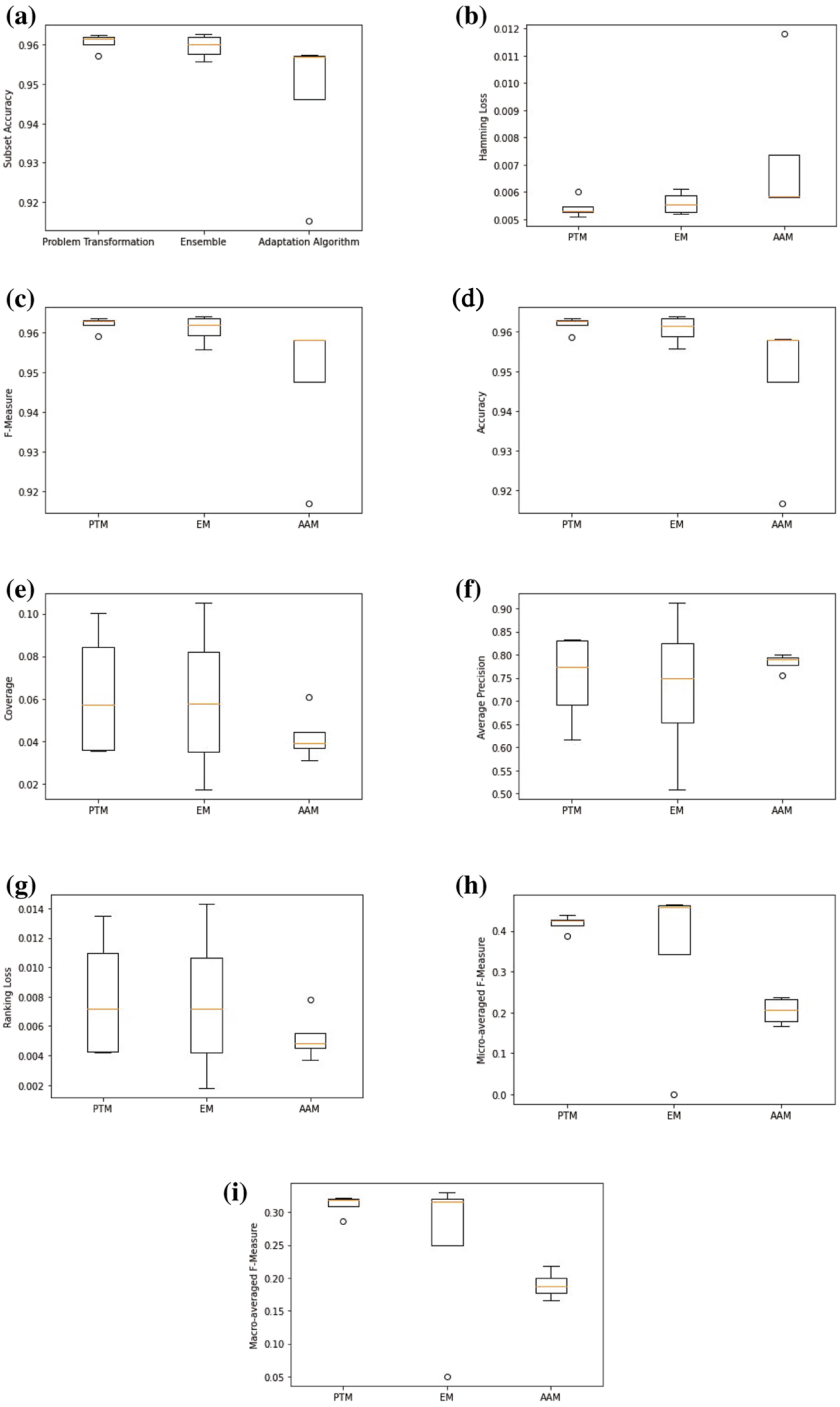

Fig. 5 Boxplots for comparing multi-label learning methods

Thus, for *RQ3*, our findings suggest that the choice of a multi-label learning method can impact the results, where problem-transformation and ensemble methods demonstrate better results in example-based classification and label-based metrics but lower results in example-based ranking metrics compared to the adaptation algorithm method.

6 Threats to validity

In this section, we discuss potential threats to the validity of our study.

- *Construct Validity:* The construction of the selected oracle combines automated and manual processes. A tool identifies potential code smells, generating a candidate list, which is then subjected to manual validation. While the presence of false positives and negatives in the oracle cannot be ruled out, it is essential to note that in the literature, this oracle is highly recognized as a well-established one for code smell identification problem.
- *Internal Validity:* In our work, the multi-label dataset exhibits a considerable Mean Imbalance Ratio, a common issue in multi-label learning due to label distribution. To address this, we implemented MLSMOTE, a Synthetic Minority Over-sampling Technique designed for multi-label learning, which has reduced the imbalance ratio.
- *External Validity:* Our experiments are carried out on 30 Java open-source projects, limiting the generalizability of our findings to other programming languages or industrial projects. Further research is needed to explore this potential limitation.

7 Conclusion

In this study, we presented a multi-label learning-based approach to identify eight class code smells across a diverse set of 30 open-source Java projects. Employing 12 different algorithms, with four selected from each of the three existing multi-label learning methods, our investigation delved into the co-occurrence of code smells at the class-level. Our analysis revealed three significant and recurring co-occurrences: {Complex Class and Large Class}, {Spaghetti Code, Large Class and Complex Class} and {Spaghetti Code and Large Class}.

We further explored the influence of correlations between various code smells on prediction outcomes. Across different evaluation metrics spanning diverse categories, we found that ECC, BR and CC emerged as the top-ranked algorithms showing that the consideration of correlations by the algorithms did not significantly impact their overall performance.

Additionally, our investigation extended to the evaluation and comparison of different multi-label learning methods, aiming to discern the efficiency of data transformation versus method adaptation in identification results. The results

suggested that the choice of a multi-label learning method can indeed impact the outcomes, where the problem-transformation and ensemble methods exhibit superior performance in example-based classification and label-based metrics but lower results in example-based ranking metrics compared to the adaptation algorithm method.

For future work, we aim to broaden our research by incorporating other types of code smells and applying the approach to different programming languages. Furthermore, we plan to utilize our research findings to create a recommendation system for prioritizing code refactoring operations.

Funding This study was not funded.

Data availability The dataset used in this study is publicly available and was obtained from the research study referenced in [3].

Declarations

Conflict of interest The authors declare that they have no Conflict of interest.

References

1. Fowler M, Beck K, Brant J, Opdyke W, Roberts D (1999) Refactoring: improving the design of existing code. Pearson Education India
2. Kaur A (2020) A systematic literature review on empirical analysis of the relationship between code smells and software quality attributes. Arch Comput Methods Eng 27(4):1267–1296
3. Palomba F, Bavota G, Di Penta M, Fasano F, Oliveto R, De Lucia A (2018) On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. Empir Softw Eng 23(3):1188–1221. https://doi.org/10.1007/s10664-017-9535-z
4. Soh Z, Yamashita A, Khomh F, Guéhéneuc YG (2016) Do code smells impact the effort of different maintenance programming activities? In: IEEE 23rd international conference on software analysis, evolution, and reengineering, vol 1, pp 393–402
5. Abbes M, Khomh F, Gueheneuc Y-G, Antoniol G (2011) An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In: 2011 15Th European conference on software maintenance and reengineering, pp 181–190. IEEE
6. Politowski C, Khomh F, Romano S, Scanniello G, Petrillo F, Guéhéneuc Y-G, Maiga A (2020) A large scale empirical study of the impact of spaghetti code and blob anti-patterns on program comprehension. Inf Softw Technol 122:106278
7. Sjøberg DI, Yamashita A, Anda BC, Mockus A, Dybå T (2012) Quantifying the effect of code smells on maintenance effort. IEEE Trans Softw Eng 39(8):1144–1156
8. Khomh F, Di Penta M, Gueheneuc Y-G (2009) An exploratory study of the impact of code smells on software change-proneness. In: 2009 16th working conference on reverse engineering, pp 75–84. IEEE
9. Cunningham W (1992) The wycash portfolio management system. ACM SIGPLAN OOPS Messenger 4(2):29–30
10. Tufano M, Palomba F, Bavota G, Oliveto R, Di Penta M, De Lucia A, Poshyvanyk D (2017) When and why your code starts to smell bad (and whether the smells go away). IEEE Trans Softw Eng 43(11):1063–1088
11. Dewangan S, Rao RS, Chowdhuri SR, Gupta M (2023) Severity classification of code smells using machine-learning methods. SN Comput Sci 4(5):564
12. Fontana FA, Zanoni M (2017) Code smell severity classification using machine learning techniques. Knowl-Based Syst 128:43–58

13. Moha N, Gueheneuc YG, Duchien L, Meur AFL (2010) DECOR: a method for the specification and detection of code and design smells. IEEE Trans Softw Eng 36(1):20–36. https://doi.org/10.1109/TSE.2009.50
14. Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A, Poshyvanyk D (2013) Detecting bad smells in source code using change history information. In: Proceedings of the 28th IEEE/ACM international conference on automated software engineering, pp 268–278. IEEE Press
15. Arcelli Fontana F, Mäntylä MV, Zanoni M, Marino A (2016) Comparing and experimenting machine learning techniques for code smell detection. Empir Softw Eng 21(3):1143–1191
16. Hadj-Kacem M, Bouassida N (2018) A hybrid approach to detect code smells using deep learning. In: Proceedings of the 13th international conference on evaluation of novel approaches to software engineering, pp 137–146. SciTePress
17. Sharma T, Efstathiou V, Louridas P, Spinellis D (2021) Code smell detection by deep direct-learning and transfer-learning. J Syst Softw 176:110936
18. Mens T, Tourwe T (2004) A survey of software refactoring. IEEE Trans Softw Eng 30(2):126–139. https://doi.org/10.1109/TSE.2004.1265817
19. Palomba F, Bavota G, Di Penta M, Fasano F, Oliveto R, De Lucia A (2018) A large-scale empirical study on the lifecycle of code smell co-occurrences. Inf Softw Technol 99:1–10
20. Tsoumakas G, Katakis I (2007) Multi-label classification: an overview. Int J Data Wareh Min (IJDWM) 3(3):1–13
21. Kreimer J (2005) Adaptive detection of design flaws. Electron Not Theor Comput Sci 141(4):117–136
22. Khomh F, Vaucher S, Guéhéneuc YG, Sahraoui H (2009) A Bayesian approach for the detection of code and design smells. In: Ninth international conference on quality software, pp 305–314 https://doi.org/10.1109/QSIC.2009.47
23. Khomh F, Vaucher S, Yann-Gaël G, Sahraoui H (2011) BDTEX: a GQM-based Bayesian approach for the detection of antipatterns. J Syst Softw 84(4):559–572
24. Hassaine S, Khomh F, Gueheneuc YG, Hamel S (2010) IDS: an immune-inspired approach for the detection of software design smells. In: Seventh international conference on the quality of information and communications technology, pp 343–348 https://doi.org/10.1109/QUATIC.2010.61
25. Oliveto R, Khomh F, Antoniol G, Gueheneuc YG (2010) Numerical signatures of antipatterns: an approach based on B-splines. In: 14th European conference on software maintenance and reengineering, pp 248–251. https://doi.org/10.1109/CSMR.2010.47
26. Maiga A, Ali N, Bhattacharya N, Sabané A, Guéhéneuc YG, Aimeur E (2012) SMURF: a SVM-based incremental anti-pattern detection approach. In: 19th working conference on reverse engineering, pp 466–475. https://doi.org/10.1109/WCRE.2012.56
27. Maiga A, Ali N, Bhattacharya N, Sabané A, Guéhéneuc YG, Antoniol G, Aïmeur E (2012) Support vector machines for anti-pattern detection. In: Proceedings of the 27th IEEE/ACM international conference on automated software engineering, pp 278–281. https://doi.org/10.1145/2351676.2351723
28. Dewangan S, Rao RS, Mishra A, Gupta M (2021) A novel approach for code smell detection: an empirical study. IEEE Access 9:162869–162883
29. Barbez A, Khomh F, Guéhéneuc Y-G (2020) A machine-learning based ensemble method for anti-patterns detection. J Syst Softw 161:110486
30. Guggulothu T, Moiz SA (2020) Code smell detection using multi-label classification approach. Softw Qual J 28(3):1063–1086
31. Kiyak EO, Birant D, Birant KU (2019) Comparison of multi-label classification algorithms for code smell detection. In: 2019 3rd international symposium on multidisciplinary studies and innovative technologies (ISMSIT), pp 1–6. IEEE
32. Boutaib S, Elarbi M, Bechikh S, Palomba F, Said LB (2022) A bi-level evolutionary approach for the multi-label detection of smelly classes. In: Proceedings of the genetic and evolutionary computation conference companion, pp 782–785
33. Li Y, Zhang X (2022) Multi-label code smell detection with hybrid model based on deep learning. In: SEKE, pp 42–47
34. Chidamber SR, Kemerer CF (1994) A metrics suite for object oriented design. IEEE Trans Softw Eng 20(6):476–493
35. Azeem MI, Palomba F, Shi L, Wang Q (2019) Machine learning techniques for code smell detection: a systematic literature review and meta-analysis. Inf Softw Technol 108:115–138
36. Aniche M (2015) Java code metrics calculator (ck). https://github.com/mauricioaniche

37. Trindade RPF, Silva Bigonha MA, Ferreira KAM (2020) Oracles of bad smells: a systematic literature review. In: Proceedings of the 34th Brazilian symposium on software engineering, pp 62–71. Association for Computing Machinery
38. Zakeri-Nasrabadi M, Parsa S, Esmaili E, Palomba F (2023) A systematic literature review on the code smells datasets and validation mechanisms. ACM J Comput Cult Herit 55(13s):1–48
39. Madeyski L, Lewowski T (2020) MLCQ: Industry-relevant code smell data set. In: Proceedings of the evaluation and assessment in software engineering. EASE '20, pp 342–347. Association for Computing Machinery. https://doi.org/10.1145/3383219.3383264
40. Read J, Pfahringer B, Holmes G, Frank E (2011) Classifier chains for multi-label classification. Mach Learn 85(3):333
41. Tsoumakas G, Katakis I, Vlahavas I (2010) Mining multi-label data. Data mining and knowledge discovery handbook, pp 667–685
42. Read J (2008) A pruned problem transformation method for multi-label classification. In: Proceedings of 2008 New Zealand computer science research student conference (NZCSRS 2008), vol 143150, p 41
43. Tsoumakas G, Katakis I, Vlahavas I (2008) Effective and efficient multilabel classification in domains with large number of labels. In: Proceedings of ECML/PKDD 2008 workshop on mining multidimensional data (MMD'08)
44. Tsoumakas G, Katakis I, Vlahavas I (2011) Random k-labelsets for multi-label classification. IEEE Trans Knowl Data Eng 23(7):1079–1089
45. Read J, Pfahringer B, Holmes G (2008) Multi-label classification using ensembles of pruned sets. In: 2008 Eighth IEEE international conference on data mining, pp 995–1000. IEEE
46. Schapire RE, Singer Y (2000) Boostexter: a boosting-based system for text categorization. Mach Learn 39(2/3):135–168
47. Zhang ML, Zhou ZH (2006) Multi-label neural networks with applications to functional genomics and text categorization. IEEE Trans on Knowl Data Eng 18:1338–1351
48. Spyromitros E, Tsoumakas G, Vlahavas I (2008) An empirical study of lazy multilabel classification algorithms. In: Proceedings of 5th hellenic conference on artificial intelligence (SETN 2008)
49. Cheng W, Hullermeier E (2009) Combining instance-based learning and logistic regression for multilabel classification. Mach Learn 76(2–3):211–225
50. Zhang M-L, Zhou Z-H (2007) ML-KNN: a lazy learning approach to multi-label learning. Pattern Recogn 40(7):2038–2048
51. Charte F, Rivera AJ, Jesus MJ, Herrera F (2015) Addressing imbalance in multilabel classification: measures and random resampling algorithms. Neurocomputing 163:3–16. https://doi.org/10.1016/j.neucom.2014.08.091
52. Charte F, Charte D (2015) Working with multilabel datasets in R: the mldr package. R J 7(2):149–162
53. Charte F, Rivera AJ, Jesus MJ, Herrera F (2015) MLSMOTE: approaching imbalanced multilabel learning through synthetic instance generation. Knowl-Based Syst 89:385–397
54. Tsoumakas G, Spyromitros-Xioufis E, Vilcek J, Vlahavas I (2011) Mulan: a java library for multi-label learning. J Mach Learn Res 12:2411–2414
55. Hall M, Frank E, Holmes G, Pfahringer B, Reutemann P, Witten IH (2009) The weka data mining software: an update. ACM SIGKDD Explor Newslett 11(1):10–18
56. Zhang M-L, Zhou Z-H (2013) A review on multi-label learning algorithms. IEEE Trans Knowl Data Eng 26(8):1819–1837
57. Gibaja E, Ventura S (2014) Multi-label learning: a review of the state of the art and ongoing research. Wiley Interdiscip Rev Data Min Knowl Discov 4(6):411–444
58. García S, Fernández A, Luengo J, Herrera F (2010) Advanced nonparametric tests for multiple comparisons in the design of experiments in computational intelligence and data mining: Experimental analysis of power. Inf Sci 180(10):2044–2064