Arabian Journal for Science and Engineering (2020) 45:2341–2369
https://doi.org/10.1007/s13369-019-04311-w

REVIEW ARTICLE - COMPUTER ENGINEERING AND COMPUTER SCIENCE

Bad Smell Detection Using Machine Learning Techniques: A Systematic Literature Review

Ahmed Al-Shaaby[1] · Hamoud Aljamaan[1] · Mohammad Alshayeb[1]

Received: 20 July 2019 / Accepted: 23 December 2019 / Published online: 7 January 2020

Abstract
Code smells are indicators of potential problems in software. They tend to have a negative impact on software quality. Several studies use machine learning techniques to detect bad smells. The objective of this study is to systematically review and analyze machine learning techniques used to detect code smells to provide interested research community with knowledge about the adopted techniques and practices for code smells detection. We use a systematic literature review approach to review studies that use machine learning techniques to detect code smells. Seventeen primary studies were identified. We found that 27 code smells were used in the identified studies; God Class and Long Method, Feature Envy, and Data Class are the most frequently detected code smells. In addition, we found that 16 machine learning algorithms were employed to detect code smells with acceptable prediction accuracy. Furthermore, we the results also indicate that support vector machine techniques were investigated the most. Moreover, we observed that J48 and Random Forest algorithms outperform the other algorithms. We also noticed that, in some cases, the use of boosting techniques on the models does not always enhance their performance. More studies are needed to consider the use of ensemble learning techniques, multiclassification, and feature selection technique for code smells detection. Thus, the application of machine learning algorithms to detect code smells in systems is still in its infancy and needs more research to facilitate the employment of machine learning algorithms in detecting code smells.

Keywords Code smell · Software quality · Machine learning · Artificial intelligent · Anti-pattern · Bad smell

1 Introduction

Code smell detection can be defined as the task of identifying potential code or design problems in a system [1–5]. Code smells occur due to programming and design mistakes caused by software developers during the software designing and programming state [2]. They can also occur for other reasons such as an incorrect analysis, incorrect integration of new models into the system, ignoring software development principles, and writing codes in a complex way [1, 2]. These smells may have a negative impact on the overall quality of the system, such as maintainability and understandability [6–10]. Therefore, the code smell detection process has motivated many researchers to propose different methods to deal with the occurrence of code smells in systems. Refactoring is proposed to alleviate and overcome code-smell-related issues. Refactoring leads to high quality, high performance, low cost, reusability, implementation and the easy development of software [1, 11].

Undertaking the code smell detection process in a manual manner is considered to be subjective. Most of these techniques mainly depend on object-oriented metrics that result in various outcomes [7]. Therefore, automated tools are proposed [12–20]. Nowadays, machine learning techniques are utilized to address code smell issues with promising results. A machine learning classifier needs first to be trained using a set of code smell examples to generate a model. The generated models are then used to identify or detect code smells in unseen or new instances. The power of the generated model

✉ Mohammad Alshayeb
alshayeb@kfupm.edu.sa

Ahmed Al-Shaaby
g201408620@kfupm.edu.sa

Hamoud Aljamaan
hjamaan@kfupm.edu.sa

[1] Information and Computer Science Department, King Fahd University of Petroleum and Minerals, Dhahran 31261, Saudi Arabia

relies on various criteria related to the dataset, the machine learning classifiers, the parameters of the classifier itself, etc.

A number of systematic literature reviews (SLRs) have been conducted in the area of bad smells. Zhang et al. [21] conducted an SLR on refactoring and code smells and reviewed papers published during the period of 2000–2009. They observed that most of the reviewed studies used a small number of code smells, and some of these smells are used by the participants in a bad way (for example, message chains). Likewise, Singh and Kaur [22] performed an SLR by reviewing 238 papers. They concentrated on the methods used to detect code smells as well the tools used for refactoring these code smells. Sharma and Spinellis [23] conducted another SLR and presented the existing knowledge associated with code smells, identified the challenges and investigated the definitions of code smells, reasons for their occurrence, their impact, and the available detection tools. Santos et al. [3] conducted an SLR to synthesize the existing knowledge on code smells. They concentrated on empirical studies that investigated how code smells affect software development. Mariani and Vergilio [24] presented an SLR of the existing research that motivates or applies search-based methods in the software refactoring activity. Their SLR was conducted by reviewing 71 primary studies. Several mechanisms have been introduced to detect code smells. Rasool and Arshad [25] performed a review on the existing tools that are used to detect code smells and report their associated challenges. Furthermore, Fernandes et al. [26] conducted a comparison between 84 smell detection tools. Fontana et al. [27] conducted a literature review that focused on code smells and automatic tools. They identified seven code smell detection tools and evaluated four of them in terms of their detection results. Garcia et al. [28] presented the code smells that are frequently recurring in software design which can have non-clear and important detrimental impact on system lifecycle properties. A recent and related SLR was undertaken by Azeem et al. [29] in which they conducted a study to provide an overview of the use of machine learning approaches for code smell detection. They identified 15 primary studies that used machine learning approaches. Their study focused on addressing four issues: (1) the code smells considered, (2) the setup of the approach, (3) the design of the evaluation strategies, and (4) an analysis of the performance. However, in this paper, we further provide a detailed analysis of the datasets used in code smell detection studies. Furthermore, we investigate the tools used to detect code smells, the tools used to extract and compute software metrics (features) and the tools used to implement machine learning techniques applied to detect code smells. Moreover, we investigate the use of feature selection techniques and compare the stand alone and ensemble-based machine learning techniques.

Another recent SLR was developed by Caram et al. [30]; the authors in this study conducted a systematic mapping study of the use of machine learning techniques for code smells identification. twenty-five primary studies were identified. Their SLR concentrated on studying code smells detection using machine learning techniques from different perspectives: (1) the detected code smells, (2) the used machine learning techniques, (3) the most used machine learning techniques for each code smell, and (4) and the performance of each of these techniques for the code smells. These perspectives are covered in this study; also we present an extensive analysis of the used datasets in the literature from different viewpoints: (1) the size of each dataset (i.e., number of included systems), (2) dependent variables, (3) independent variables, (4) the used tools to compute or extract metrics (independent variables), (5) the used tools to assign dependent variables(smelly or not smelly), and (6) description analysis of each available dataset (i.e., number of features, instances and smelly, and non-smelly instances in each dataset). Further, we explore the setup of used techniques such as, the used evaluation metrics, the used validation methods, and classification type (i.e., binary classification or multiclassification). Furthermore, we investigate the use of ensemble machine learning techniques and compare them with stand-alone machine learning techniques. Moreover, we show the tools used to implement machine learning techniques. In addition, we investigate the use of feature selection techniques.

In this study, our main objective is to systematically review the studies carried out to detect code smells using machine learning algorithms from different perspectives. To achieve the study objectives, we carried out an SLR following the general guidelines defined by Kitchenham and Charters [31]. First, a wide literature search was conducted in five online databases to identify the relevant studies. Then, a set of inclusion and exclusion criteria and quality assessments were devised to obtain the primary studies. The selected studies were then analyzed, classified, and compared using our defined criteria including types of machine learning algorithms, prediction accuracy, detected code smells and datasets, resulting in the selection of 17 primary studies.

The results of this study provide knowledge for both practitioners and researchers about the most frequently code smells detected, and the machine learning technique used to detect them. The results also provide information about the accuracy measures used in the experimental studies that practitioners and researchers can use for comparison with the existing studies. The study also provides analysis about the tools used for code smell detection and correction. Furthermore, the details of the datasets used in bad smell detection and correction studies are also reported.

The main contributions of this study are:

1. Identified 17 primary studies that use machine learning techniques to detect code smells.

2. Conducted different analyses on these primary studies to provide knowledge about: (1) the applied techniques, (2) the detected code smells, (3) the accuracy measures, (4) the used datasets, and (5) the most commonly used tools.
3. Provide recommendations that can be used for the future research.

The rest of this study is organized as follows. A background of bad smells and machine learning techniques is presented in Sect. 2. The research methodology used is discussed in Sect. 3. Section 4 presents the results. Section 5 discusses the main findings. In Sect. 6, we identify the potential threats to the validity of our study, while Sect. 7 presents the conclusion.

2 Background

In this section, we provide a brief background of code smells and machine learning techniques.

2.1 Code Bad Smells

Code smells indicate potential code or design problems in a system [1, 2]. Code smells, also known as design flaws, refer to design situations that negatively influence the maintainability of the software [27]; therefore, they may impact the maintenance processes [32]. A number of code smells were presented in Fowler's book [33]. Fowler also suggested guidelines to eliminate these smells from the system [25].

It is helpful if we identify a bad smell as early as possible in the development lifecycle [32, 34]. Detecting bad smells in code or in the design and then performing the appropriate refactoring procedures when necessary is very useful to enhance the quality of the code. These smells make the system more difficult to maintain, probably also increasing its fault proneness [35]. Bad smells are unlikely to result in failure directly but might do so indirectly which still negatively impacts software quality [36].

Code metrics are used by different code smell detection tools to detect code smells. These tools either compute the metrics from the code itself or utilize the extracted metrics from the external party's tools [7]. The bad smell detection process can either be manual or automatic, using detection strategies. Bad smells can be detected in source code or system design.

2.1.1 Code smell categories

There are several categories of bad smells, and each category contains several types. In our study, we selected the category proposed in [33, 37–39], as they include the most common code smells types. The code bad smells are very closely relevant. Consequently, we consider that the taxonomy makes the smells more recognizable and understandable. Figure 1 details these categories.

- *Bloaters* In this category, the code or classes are expanded to such a large extent that they are difficult to work with. These smells do not manifest immediately, rather they aggregate after some time as the program develops, particularly when no one endeavors to eliminate them. The first type of code smell in this category is the Long Method, which contains too many lines of code, making it difficult to reuse, change, and understand. The best solution for this smell is to divide this method into separate methods. The second type of code smell in this category is Large Class. This occurs when a single class attempts to do too much, and it usually contains several instances and has various responsibilities. This smell makes the reusability and maintainability of this class more difficult. The best solution for this smell is splitting this class, by applying extract class. The third type of code smell in this category is Primitive Obsession used in software. We should use small classes instead of primitive types in some situations. For instance, primitives are used in place of small objects for simple tasks, for example, special strings for phone numbers, ranges, and currency. The fourth type of code smell in this category is Long Parameter List. This type of smell occurs if any method has more than four parameters, making parameter lists more difficult to understand and use and also inconsistent. The final type of code smell in this category is Data Clumps. Occasionally, various parts of the code contain identical groups of variables, for example, parameters to connect to a database. These clumps should be turned into their own classes.
- *Object-Orientation Abusers* All the smells in this category involve the incomplete or incorrect application of the principles of object-oriented programming. The first type of bad smell in this category is the Switch Statements. This smell appears in the code when it contains a sequence of *if* statements or a complex switch operator. The second type of bad smell in this category is temporary field. Usually, temporary fields are created for use in an algorithm that requires a lot of parameters. Therefore, instead of creating a large number of parameters, the programmer creates fields for these data in the class. These fields are utilized only in the algorithm and go unused the rest of the time. This kind of smell is difficult to discover. Removing this smell enhances code clarity and organization. The third type of bad smell in this category is Refused Bequest. This smell occurs when programmers create inheritance between two completely different classes, but the subclass uses only a few of the methods

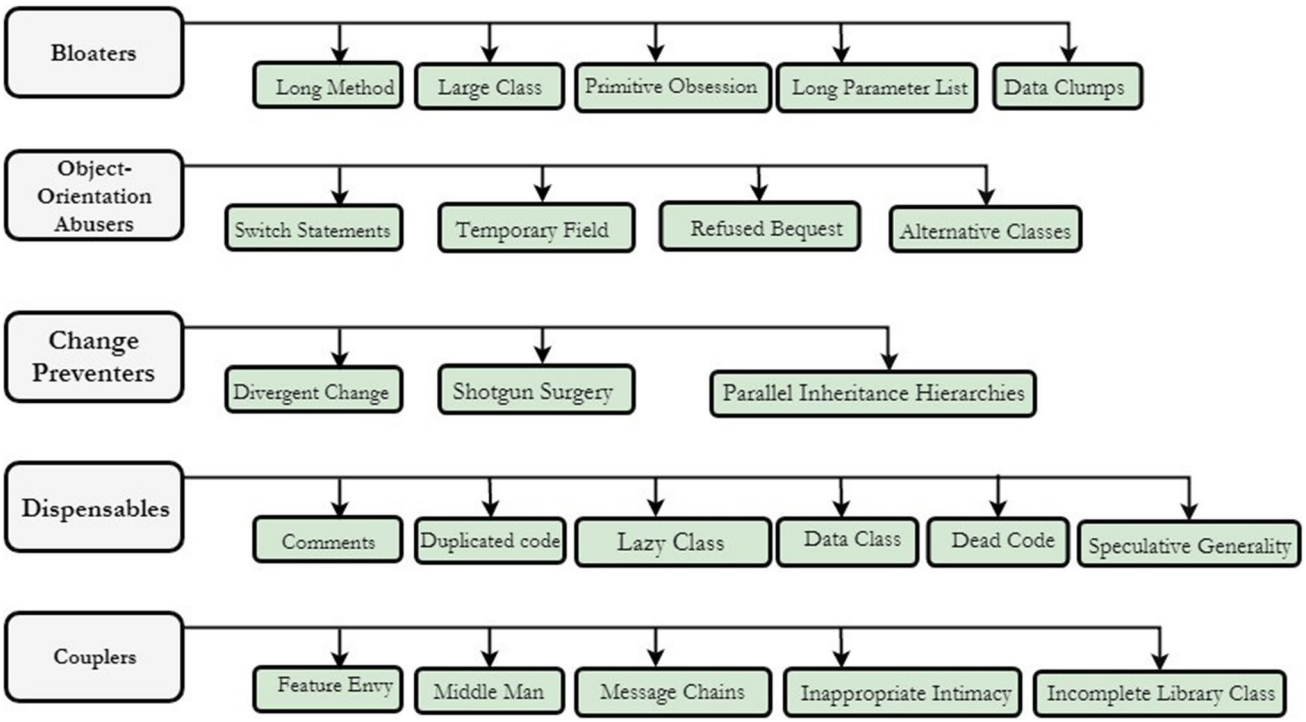

Fig. 1 Code smell categories

and properties inherited from the superclass. The best way to treat this smell is to use delegation instead of inheritance. The fourth type of bad smell in this category is alternative classes with different interfaces. This code smell occurs when programmers create two classes with identical functionality, but these methods have different names.

- *Change Preventers* These smells occur if you need to modify something in one place in your code, and then you have to make many modifications in other places too. Therefore, the development process of the software is more complicated and costly. In this category, there are three types of bad smells. The first type of bad smell in this category is Divergent Change. This bad smell occurs when many changes are made to a single class. The best way to remove this smell is to split the class's behavior. For instance, in the case where different classes have the same behavior, the classes should be combined through inheritance. This will improve the organization of the code as well as reduce code duplication. The second type of bad smell in this category is Shotgun Surgery. This occurs when a single change is made to multiple classes simultaneously. The reason this smell occurs is because one responsibility has been split up among a large number of classes. The best way to remove this smell is to move the existing class behaviors into a single class. This will improve the organization of the code, reduce code duplication, and make it easier to maintain. The third type of bad smell in this category is Parallel Inheritance Hierarchies. This smell occurs when you create a subclass for a class and then discover that you need to create a subclass for another class.
- *Dispensables* These smells occur when part of the code is not needed and where it to be removed, the code would be cleaner, more efficient and easier to understand. There are six types of bad smells in this category. The first type of bad smell in this category is Comments. This smell occurs when the program is filled with explanatory comments. The second type of bad smell in this category is the Duplicate Code. This smell occurs when the same or very similar code appears in several parts of the program, making the program code large. This bad smell can be removed by creating a new method that encapsulates the duplicated code. The third type of bad smell in this category is Lazy Class which is a useless class. Every class which is built takes effort and is time-consuming to understand and maintain. The best way to remove this smell is to eliminate these classes. The fourth type of bad smell in this category is Data Class. This is a class that contains only fields but there is seldom any logic to it. The Data Class has getters and setters methods for fields. The fifth type of bad smell in this category is Dead Code. This occurs when a code is never executed. The sixth type of bad smell in this category is Speculative Generality. This occurs when there is an unused parameter, field, method, or class. The reason this bad smell occurs

is because sometimes, code is created to support anticipated future features that are never actually implemented. Consequently, the code becomes difficult to understand and support.

- *Couplers* All the smells in this category contribute to an excessive coupling between classes or show what happens if coupling is replaced by excessive delegation. There are six types of bad smells in this category. The first type of bad smell in this category is Feature Envy. This bad smell occurs a method accesses the data of another object more than its own data. It generally occurs when fields are moved to a Data Class. If this happens, the operations on data should be moved to this class as well. The second type of bad smell in this category is Inappropriate Intimacy. This occurs when one class uses the internal methods and fields of another class to do its work. The third type of bad smell in this category is Message Chains. This occurs when a client requests another object, and that object requests yet another object and so on. The fourth type of bad smell in this category is Middle Man. This occurs if a class performs only one action and delegates work to another class, hence there is little point in its existing. This smell can be the result of the overzealous elimination of Message Chains. The fifth type of bad smell in this category is Incomplete Library Class. This occurs when libraries no longer meet user requirements.

2.2 Machine learning

Machine learning is a discipline where computer systems are able to learn and perform their work even if they were not explicitly programmed [40]. The most commonly used machine learning techniques in the literature on software quality prediction are supervised learning, reinforcement learning, and unsupervised learning. This subsection describes the investigated classification models.

- *Multilayer Perceptron* (*MLP*) [41, 42] This is an artificial neural network (ANN) model that consists of a layer as an input, at least one hidden layer, and an output layer. Every node is a neuron that utilizes a nonlinear activation function, and it associates with other nodes in the next layer with a specific weight. MLP constantly utilizes the backpropagation technique for training.
- *Support Vector Machines* (*SVMs*) [43, 44] These are supervised learning models with related learning techniques, and these learning algorithms can be used for regression or classification. SVM was defined by Vapnik [45] based on the principle of structured risk minimization. The main objective of SVM is empirical error minimization and geometric margin maximization. Commonly, the extent to which deviations are tolerated, the complexity parameter C, and the kernel are the parameters that are used to define the SVM model.
- *Radial Basis Function Networks* (*RBFs*) [46, 47] This is a type of neural network that has three layers, one being an input layer, the second being a hidden layer, and the third being a linear output layer. Three types of RBFs are multiquadric, polyharmonic spline, and Gaussian. RBF networks are used for classification, function approximation, and system control.
- *Bayesian Belief Networks* (*BBNs*) [48] This is a convenient graphical model for representing a collection of variables and their probabilistic independencies. In this model, a random variable is represented by a node in the graph, whereas the probabilistic dependencies among the corresponding random variables are represented by the edges connecting the nodes.
- *Naive Bayes* (*NB*) [47, 49] This is a supervised learning algorithm that employs the Bayes algorithm with the "naive" assumption of conditional independence among each pair of attributes.
- *Random Forests* (*RF*) is a supervised learning algorithm that contains many unpruned classifications or regression trees such that every tree is based on the values of a random vector experimented individually and with the same distribution for all trees in the forest. RF can be employed for both classification and regression problems [50, 51].
- *Linear Regression* (*LR*) is a modeling method that is applied to find the correlation between the target and independent variables in the dataset by utilizing the linear predictor functions [52–54].
- *Multinomial Naive Bayes* (*MNB*) Multinomial naive Bayes is a version of NB that is introduced for text classification. In MNB, the data samples follow a multinomial distribution [49, 55].
- *Decision Tree* (*DT*) [56] This is one of the most successful options for the supervised learning method for regression and classification. The C4.5 algorithm is the most commonly used technique to generate decision trees [57].

3 Research Methodology

The main objective of this study is to identify and analyze all relevant studies that use machine learning to detect code smells. As previously mentioned, we followed the SLR guidelines suggested by Kitchenham and Charters [31]. An SLR is a well-defined and systematic way of finding, assessing, and analyzing published primary research [58–61]. The SLR gives a strong basis on which to make claims on research questions, but it needs considerably more effort than a traditional literature review [62]. The SLR process involves six phases, followed in sequence as shown in Fig. 2:

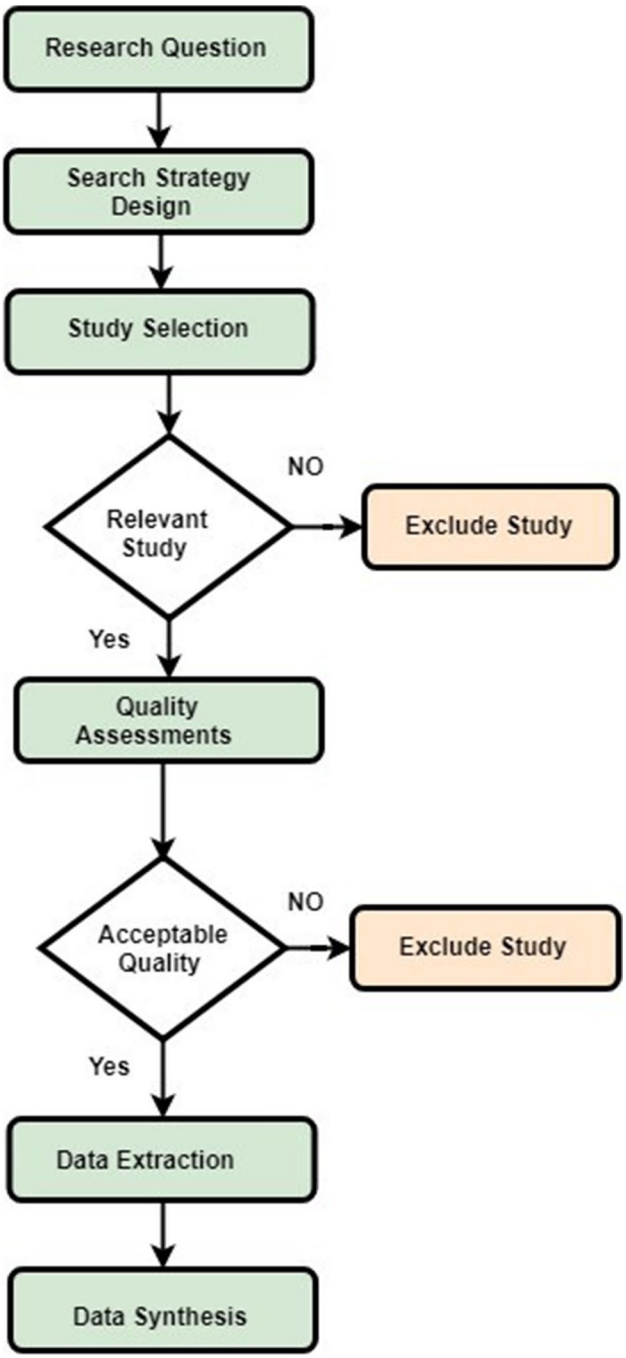

Fig. 2 Systematic literature review process phases

- Research questions
- Search strategy design
- Study selection
- Quality assessment
- Data extraction
- Data synthesis

In the first phase, a number of research questions based on the goal of the current SLR are defined. Then, a search strategy is built to find all the studies relevant to the research questions. Next, the search string, the digital libraries and the inclusion and the exclusion criteria are identified, and in the fourth phase, a set of quality assessment criteria are identified and applied to the selected papers. In the data extraction phase, the data extraction cards are created and employed to obtain data from the selected papers. Finally, in the data synthesis phase, appropriate methodologies are defined to synthesize the extracted data.

3.1 Research questions

Constructing the research questions is a significant step in the SLR process. Five research questions are defined to achieve our research objective:

RQ1: Which machine learning techniques have been applied to detect code smells?

The objective of RQ1 is to identify the machine learning techniques that have been applied to detect code smells. Researchers can use the outcomes of this question to identify the most applied machine learning techniques for code smells detection and to investigate the possibility of implementing unused techniques.

RQ2: Which code smells are most commonly detected using machine learning techniques?

Several studies have been conducted to address the issue of code smells, so our objective in relation to this question is to identify the code smells that have been detected using machine learning and why researchers have chosen these code smells. The findings of this question can be used to determine the code smells that have not been investigated yet or got less attention by current studies. Hence, researchers can address them in the future work.

RQ3: What are the accuracy measures of the machine learning techniques that have been used for code smell detection?

To answer this question, we identify the performance metrics used to evaluate the machine learning techniques in terms of the detection of code smells. Then, based on these performance metrics, we identify the accuracy of the machine learning techniques that have been used for code smell detection. Next, we compare the machine learning techniques that have been used to detect code smells in order to find the most efficient. The findings of this question can be useful to identify the accuracy measures used to evaluate the

performance of the used machine learning techniques. Such knowledge enables researches to use the most appropriate accuracy measure in their studies.

RQ4: What datasets have been used for code smell detection?

The objective of this question is to investigate the attributes of these datasets, such as: the dataset name, size (number of systems in each dataset, the size of each system), type (commercial, student, open source), availability of the datasets (available online or not), the language of the selected systems, the inputs of the datasets, the tools used to obtain the values of dataset inputs, and an analysis of the available datasets. Researchers can use the outcomes of this question to build their datasets in a good way.

RQ5: What are the most commonly used machine learning tools in the context of bad smell detection?

Many tools are used to implement machine learning algorithms. To answer this question, we investigate the tools that are used to implement machine learning algorithms and explain why they have been selected. The findings of this question can be useful to identify the most used tool; thus, researchers can select the most suitable tool for their needs.

3.2 Search strategy

This involves three phases: search terms, online databases, and the search process. These are described in the following subsections.

3.2.1 Search strategy

The common strategies applied to construct the search terms are described in this subsection as follows:

1. Obtaining the main terms from the research questions.
2. Finding the alternate synonyms and spelling for the main terms.
3. Verifying the above steps by matching the keywords from any relevant research paper.
4. Managing the Boolean operator "OR" to link the alternative synonyms and spellings and "AND" to link the major terms and the Boolean operator.

The search terms are built based on population, intervention, outcome, and experimental design.

- *Population* Code smells.
- *Intervention* The existing machine learning techniques for detecting code smells.
- *Outcomes* Improve software quality.
- *Experimental Design* Empirical studies, case studies, and experimental studies.

After applying the previous steps along with several tests results, the following complete search terms are employed in our research.

((((Code OR Bad) AND Smell*) OR Antipatterns OR Refactoring) **AND** (Detect* OR Predict* OR Estimat* OR Forecast*) **AND** (("Machine learning" AND (Model OR Technique OR Algorithm OR Method OR approach)) OR "artificial intelligence" OR "Ensemble learning")).

Table 1 Online databases

Number	Name	URL
1	IEEE Xplore	http://ieeexplore.ieee.org
2	Springer Link	http://link.springer.com
3	Science Direct	http://www.sciencedirect.com
4	Scopus	https://www.scopus.com
5	ACM Digital Library	http://dl.acm.org

3.2.2 Research Resources

Five online databases are used to find relevant conference and journal papers using our defined search terms. Table 1 presents these online databases. These databases were selected as they are the popular venues for publishing papers on machine learning and bad smell detection studies. Other researchers have also used these databases in their SLR studies [3, 63–65].

The search terms are modified to be compatible with each online database since each has its own search engine syntax. Furthermore, the grammars for searching differ from one to another. The start date of the searching process was open until December 2018.

In order to collect relevant material avoiding bias, a broad range of research databases are considered which includes all journal papers and conference papers.

3.2.3 Search Process

In general, the SLR process involves a comprehensive search of all the studies that fit the selection criteria. The search process comprises the following two phases:

- *Phase* 1 The five online databases are searched separately using the constructed search string. The identified studies are then collected to establish a set of candidate studies. Table 2 presents the results of this phase.

Table 2 Number of collected studies

Name	#Collected studies	Selection criteria results
IEEE Xplore	147	20
Springer Link	247	5
Science Direct	166	5
Scopus	169	20
ACM Digital Library	700	6
Total	1429	56

- *Phase* 2 For each relevant study, the reference lists are scanned to find other relevant studies to be included into our candidate list.

It is expected that many references will be collected during the whole SLR process. EndNote is used to keep a record of all the references. Consequently, 1429 relevant studies are retrieved.

3.3 Study Selection

The inclusion and exclusion criteria are defined in this subsection. A total of 1429 candidate studies are retrieved as described in Table 2 and Fig. 3. Most of them do not provide valuable information to fit the defined research questions. Therefore, more filtering is recommended in order to identify relevant studies. The study selection process is illustrated in Fig. 3. The selection process is performed by one researcher, and another two researchers verify the selection process at random. The selection process is carried out in two phases:

- *Initial Selection* Studies found during the initial search are assessed for relevance. This is done by analyzing the title and abstract as recommended by [66] and by applying the inclusion and exclusion criteria (defined below) to identify the relevant studies, which provide information to answer the research questions.
- *Final Selection* Studies found in the initial selection underwent further analysis (i.e., reading full text), and the quality assessment criteria were applied. This is done so that no important information or data with respect to the research question are missed. These studies will be eventually used for data extraction.

3.3.1 Inclusion Criteria

For a paper to be included in the SLR, it needs to meet various inclusion criteria.

- Studies that propose and discuss the use of machine learning techniques to detect code smells.
- Studies that motivate and describe the benefits of using machine learning techniques to detect code smells.
- Studies that provide an empirical basis for their findings.
- In the case of duplicate studies, only the most complete and recent will be selected.
- Papers which have been published in a journal or in conference proceedings.

3.3.2 Exclusion Criteria

Several exclusion criteria were established to make certain studies ineligible for inclusion in the SLR.

- Studies that are not applicable to the research questions.
- Studies that do not use or propose machine learning techniques to detect code smells.
- Studies that are not written in the English language.
- Publications that do not have an empirical analysis or findings from applying machine learning models to detect code smells.

As a result, 56 relevant studies were obtained, as described in Table 2. Next, we scanned the references in these relevant studies, but we did not find any additional relevant studies.

3.4 Study Quality Assessments

After applying the selection criteria, we define the quality assessment criteria for the selected papers according to the objectives of this research. These quality assessments are used to weight the selected studies. In this subsection, we list the quality assessment questions used for quantitative assessment of the quality of the selected papers where Yes = 1, No = 0, and partly = 0.5. The quality assessment questions are listed in Table 3. The final score is computed by summing the values assigned to each question, where the higher the score, the higher the quality of the study. In our research, we selected only those papers with a quality score greater than 4.5. We used the same threshold (50%) that was used in [67]. Two researchers conducted these quality assessments of the selected studies separately. In the case of disagreement between the researchers, it was discussed and resolved. Table 4 shows the quality scores of the selected studies. Seventeen studies were identified as the final papers for the data extraction process. These 17 studies are described in Table 5.

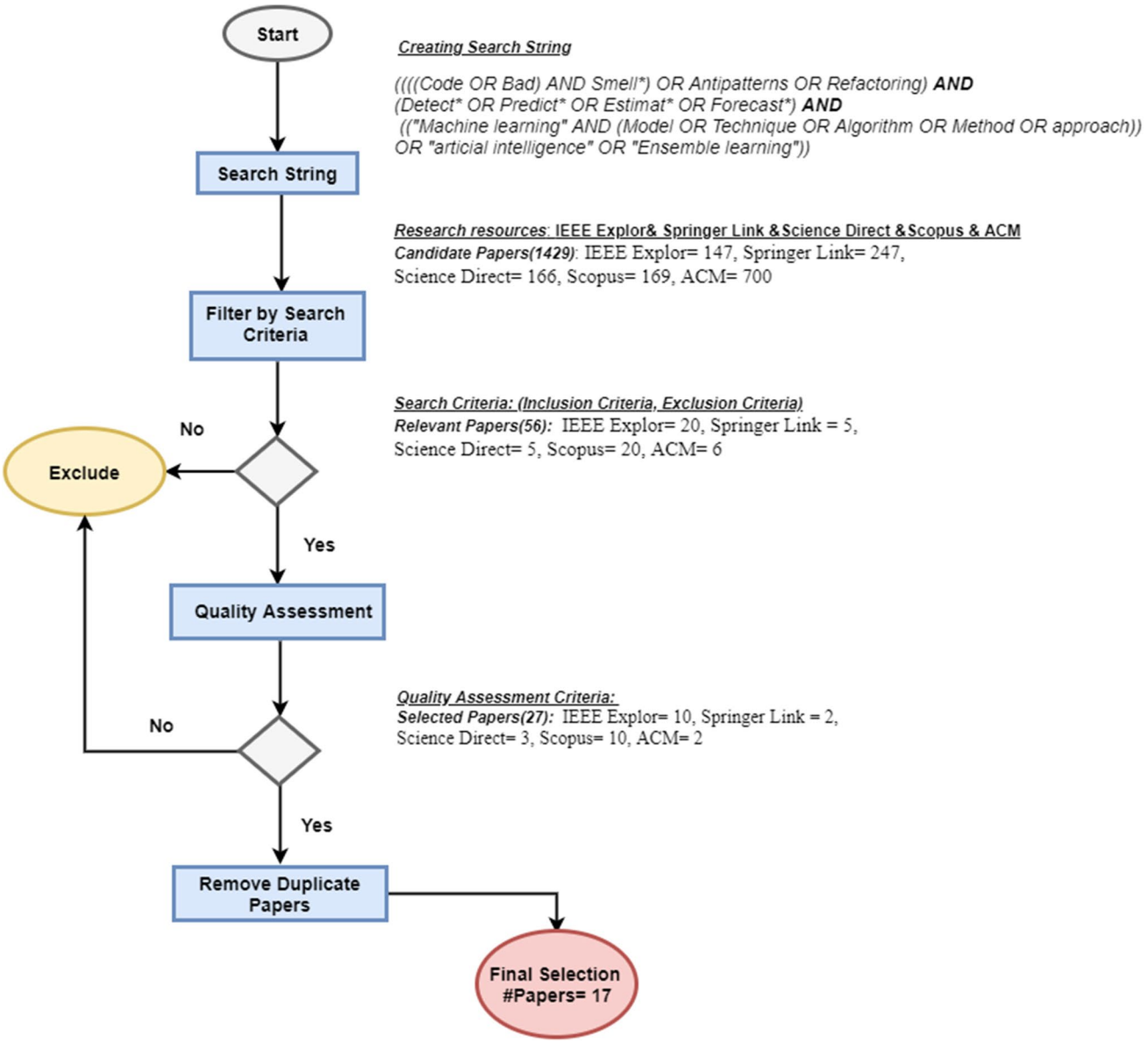

Fig. 3 Selection process phase

Table 3 Quality assessment questions

Q#	Quality questions	Yes	No	Partly
Q1	Are the research objectives clearly stated and well motivated?			
Q2	Is the experiment design clearly defined?			
Q3	Are the performance measures used to measure the machine learning models clearly explained?			
Q4	Is the dataset size sufficiently stated?			
Q5	Are the detected code smells clearly defined?			
Q6	Are the machine learning models employed sufficiently defined?			
Q7	Are the independent variables clearly defined?			
Q8	Are the findings of the study sufficiently defined well motivated?			
Q9	Does the study discuss threats to validity or limitations?			

Table 4 Quality scores of selected studies

Study	Reference	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Score
S1	[68]	1	1	1	1	1	0.5	1	1	1	8.5
S2	[69]	1	1	0.5	1	1	0.5	1	0.5	0	6.5
S3	[70]	1	1	1	0.5	1	1	0.5	0.5	0.5	7
S4	[7]	1	1	1	1	1	1	0.5	1	1	8.5
S5	[52]	1	1	0.5	0.5	0	1	1	0.5	0	5.5
S6	[71]	1	1	1	0.5	1	1	1	0.5	0	7
S7	[72]	1	1	1	0.5	1	1	1	0.5	0	7
S8	[73]	1	1	1	1	1	1	1	0.5	1	8.5
S9	[74]	1	1	1	1	1	1	0.5	0.5	0.5	7.5
S10	[32]	1	1	1	1	1	1	0.5	0	0	6.5
S11	[75]	1	1	0.5	0	1	0.5	0	0.5	0.5	5
S12	[6]	1	1	1	0.5	1	0.5	0.5	0..5	0	6
S13	[76]	1	1	0.5	0.5	1	0.5	0.5	0.5	0	5.5
S14	[77]	1	1	1	0.5	1	1	0.5	0.5	0.5	7
S15	[78]	1	1	0.5	0.5	1	1	1	0.5	0.5	7
S16	[79]	1	0.5	1	0.5	1	1	0.5	0.5	0.5	6.5
S17	[80]	1	1	0.5	0.5	0	1	1	0.5	0	5.5

Table 5 Selected primary studies

#	Type	Online database	Year	Reference
S1	Journal	Scopus& Science Direct	2018	[68]
S2	Journal	Scopus	2017	[69]
S3	Conference	Scopus & IEEE	2017	[70]
S4	Journal	Springer Link	2016	[7]
S5	Conference	Scopus & IEEE& Science Direct	2016	[52]
S6	Conference	Scopus & IEEE	2015	[71]
S7	Conference	Scopus & IEEE	2013	[72]
S8	Conference	IEEE & ACM& Digital Library	2012	[73]
S9	Journal	Science Direct	2011	[74]
S10	Conference	Scopus & IEEE	2011	[32]
S11	Conference	Scopus & IEEE	2010	[75]
S12	Conference	Scopus & IEEE	2009	[6]
S13	Conference	Scopus	2005	[76]
S14	Journal	Springer Link	2015	[77]
S15	Conference	ACM	2012	[78]
S16	Conference	IEEE	2010	[79]
S17	Conference	IEEE	2009	[80]

3.5 Data Extraction

The selected papers are used to gather data that address our research questions. In order to make this process easier, cards are created with the form described in Table 6. This form is refined by pilot data extraction with many of the selected papers. To make the process of data synthesis as easy as possible, we group the items based on the association between the research questions, as shown in Table 6.

Table 7 demonstrates that the data extracted to address **RQ3**, **RQ4**, and **RQ5** are associated with the experiments conducted in the chosen papers. Generally, in the software engineering field, the word experiment has diverse meanings. Consequently, to avoid confusion, we explicitly define the word "experiment" as a process in which an algorithm or a model is assessed with suitable performance metrics depending on a specific dataset.

The data extraction cards were employed to obtain data from the chosen papers. Three researchers undertook this process. One extracted the data and then inserted these data into the cards, while the others examined the collected data. If there was disagreement between the researchers about the results, a discussion was conducted to resolve the conflict. Finally, the examined extracted data are documented in a file, to be utilized in the next process.

As shown in Table 7, all our research questions were answered by the primary studies except five studies (**S5, S11, S14, S16,** and **S17**) which did not address **RQ5** because they did not mention the tools used to implement the machine learning techniques. To facilitate the tracking of the extracted data, we marked every paper with an ID, Table 7 presents more detail.

3.6 Data Synthesis

Data synthesis aims at aggregating the extracted data from the selected papers to answer our research questions. In this study, the data extracted comprise qualitative data (e.g., a list of the employed machine learning techniques, the size of each dataset, and a list of the detected code smells) and quantitative data (e.g., the values of the prediction accuracy).

Table 6 Data extraction card

Name of extractor
Date of extraction
Data checker
Paper ID
Title of paper
Authors name
Publication year
Source
Reference type (Journal/Conference)
Study type (experiment, case study) publisher
RQ1: Which machine learning techniques have been applied to detect code smells?
Machine learning techniques applied to detect code smells.
RQ2: Which code smells are most commonly detected using machine learning techniques?
The selected code smells in the field of using machine learning techniques to detect code smells, and why they have selected these smells?
RQ3: What are the accuracy measures of the machine learning techniques that have been used for code smell detection?
Metrics employed to estimate the detection performance, detection accuracy values and validation mechanism.
RQ4: What datasets have been used for code smell detection?
Datasets used in the experiments. Dataset name, size, type (commercial, student, open source), availability, language, metrics used, and tools used to compute metric value.
RQ5: What are the most commonly used machine learning tools used in the context of bad smell detection?
Tools used to implement machine learning techniques to detect code smells.

Table 7 Addressed questions

ID	Authors	The addressed research questions				
S1	D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia.	1	2	3	4	5
S2	D. K. Kim.	1	2	3	4	5
S3	A. Kaur, S. Jain, and S. Goel.	1	2	3	4	5
S4	F.A. Fontana, M. V. Mantyla, M. Zanoni, and A. Marino.	1	2	3	4	5
S5	Tarwani, Sandhya, and Anuradha Chug.	1	2	3	4	–
S6	L. Amorim, E. Costa, N. Antunes, B.Fonseca, and M. Ribeiro.	1	2	3	4	5
S7	F. A. Fontana, M. Zanoni, A. Marino, and M. V. Mantyla.	1	2	3	4	5
S8	A. Maiga, N. Ali, N. Bhattacharya, A. Sabane.	1	2	3	4	5
S9	F. Khomh, S. Vaucher, Y.-G. Gueheneuc, and H. Sahraoui.	1	2	3	4	5
S10	N. Maneerat and P. Muenchaisri.	1	2	3	4	5
S11	S. Bryton, F. B. e Abreu, and M. Monteiro.	1	2	3	4	–
S12	F. Khomh, S. Vaucher, Y.-G. Gueheneuc, and H. Sahraoui.	1	2	3	4	5
S13	J. Kreimer.	1	2	3	4	5
S14	Yang et al.	1	2	3	4	–
S15	Wang et al.	1	2	3	4	5
S16	Hassaine et al.	1	2	3	4	–
S17	Vaucher et al.	1	2	3	4	–

As only 17 papers were selected, we employed canonical meta-analysis [31] as the synthesis methodology because of its solid theoretical background. Employing standard meta-analysis with this number of studies would be inappropriate [31].

To synthesize the extracted data relating to various research questions, we set the narrative synthesis strategy for **RQ1**, **RQ2**, and **RQ5**. We tabulated the data in a style compatible with the questions.

4 Results

In this section, we present the main results of this review. First, a summary of the selected papers is presented. Then, we document the main findings of our work based on the research questions.

Table 8 Distribution of publication venues

Publication type	# of studies
Conference	12
Journal	5

4.1 Overview of Selected Studies

The issue of detecting code smells [81] has drawn the attention of many researchers over recent years [26]. The research literature can be categorized into two: empirical studies and prediction models. Empirical studies conduct research with the intention of knowing code smell growth [82–85], their perception [86–88], and their influence on the quality attributes of source code [89–91]. Studies on prediction models depend on the analysis of structural information of the code itself [14, 15, 92]. The recent studies focus on the analysis of other sources of information [93, 94] or the utilization of software engineering techniques which are search based [21, 36]. In the context of our research, we concentrate on machine learning techniques for detecting bad smells. Therefore, we review studies which utilize machine learning techniques to predict bad smells.

In our study, we identified 17 *primary studies (PSs), listed in Table 4, that relate to bad smell detection using machine learning techniques, published between* 2005 *and* 2018.

As shown in Table 8, five of the PSs were published as journal articles, and 12 were published in conference proceedings. The distribution of the publication venues is presented in Tables 4 and 8. All the selected studies are experiment papers. The number of PSs published per year is shown in Fig. 4. It can be observed that the number of studies published between 2009 and 2012, and between 2015 and 2017 remained stable with two studies published per year, whereas in 2005, 2013, and 2018, one study was published each year. Furthermore, there is a slight increase in the number of published studies in this area in recent years, which indicates that research attention in this area is growing.

4.2 Types of machine learning techniques used (RQ1)

Eighteen types of machine learning algorithms were employed to detect code smells. Table 9 presents our findings. We found that 12 of our PSs applied a single machine learning technique per code smell detection, while the remaining 5 studies applied more than one machine learning technique. The table presents the machine learning techniques used in descending order from the highest frequency to the lowest. Study **S10** used seven machine learning techniques which was the highest.

We observe that some of the applied machine learning techniques are related to the same family, for example, J48 and Decision Trees C5.0 are from the same decision tree family, as shown in Table 10; the family of the decision tree is the most commonly applied to detect code smells. In addition, we found that the authors of studies **S1, S4,** and **S7** used the same machine learning techniques. The authors of **S4** and **S7** are the same, whereas the authors of study **S1** replicated study **S4**.

The process of identifying the machine learning methods that were used to detect code smells is based on the general name of the machine learning technique. For example, in the studies of [7, 68], they used SVM and sequential minimal optimization (SMO) with different kernels and they also used boosting techniques for all the used techniques, as shown in Table 11. We identify only the general name of the techniques, so we identify six machine learning techniques

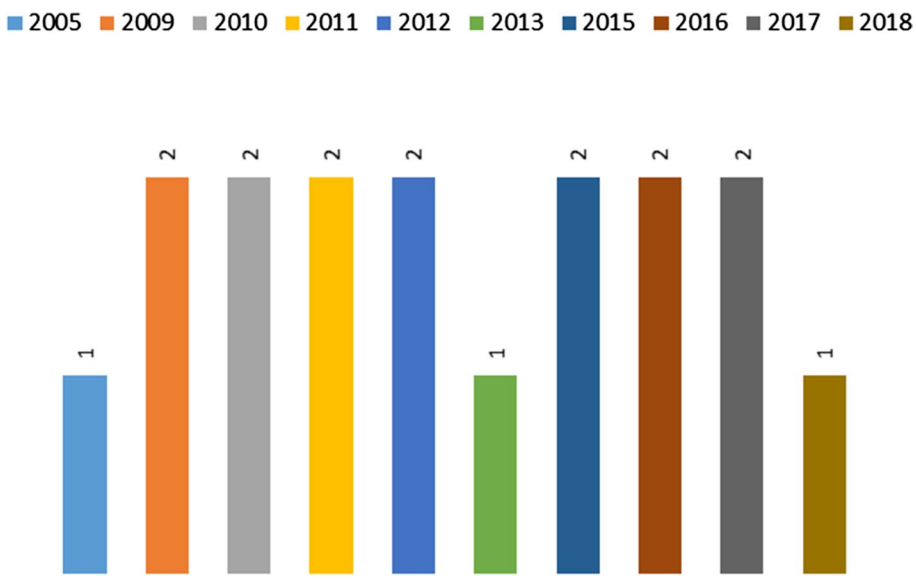

Fig. 4 Number of PSs published per year

Table 9 Distribution of machine learning techniques in the selected studies

Used ML/studies	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15	S16	S17	Count	Frequency %
Support vector machines (SVM)	√		√	√	√		√	√										6	35.29
J48	√			√			√			√								4	23.52
Naive Bayes	√			√			√			√								4	23.52
Random Forest (RF)	√			√			√			√								4	23.52
Bayesian Belief Networks (BBN)									√			√			√		√	4	23.52
JRIP	√			√			√											3	17.64
Sequential minimal Optimization (SMO)	√			√			√											3	17.64
Decision Trees C5.0						√							√					2	11.76
Binary Logistic Regression (BLR)										√	√							2	11.76
Multilayer Perceptron (MLP)		√			√													2	11.76
Radial Basis Function Neural Networks (RBFN)					√													1	5.88
Decision tree Forest (DFT)					√													1	5.88
Linear Regression (LR)					√													1	5.88
Instance-Based Learning with Parameter k (IBK)										√								1	5.88
Voting Feature Intervals (VFI)										√								1	5.88
Instance-Based Learning (IBL)										√								1	5.88
Natural Language Processing														√				1	5.88
Artificial Immune Systems (AIS)																√		1	5.88
#Used machine learning techniques	6	1	1	6	5	1	6	1	1	7	1	1	1	1	1	1	1		

Table 10 Machine learning family

Technique family	Techniques
Artificial Neural Network	RBFN, MLP
Decision Tree	J48, Decision trees C5.0, Decision tree forest (DFT), Random Forest
Support Vector Machines	SVM, SMO
Instant-Based Learning	IBL, IBK

Table 11 The ML techniques used

Machine learning techniques used in the selected studies	Study
B-J48 Pruned, B-J48 Unpruned, B-J48 Reduce Error Pruning, B-JRip, Random Forest, B-Random Forest, Naive Bayes, B-Naive Bayes, B-SMO RBF Kernel, B-SMO Poly Kernel, B-LibSVM C-SVC Poly Kernel, B-LibSVM C-SVC Linear Kernel, B-LibSVM C-SVC Radial Kernel, B-LibSVM C-SVC Sigmoid Kernel, J48 Pruned, J48 Unpruned, J48 Reduce Error Pruning, JRip, SMO Polynomial, SMO RBF, LibSVM C-SVC Poly Kernel, LibSVM C-SVC Linear Kernel, LibSVM C-SVC Radial Kernel, LibSVM C-SVC Sigmoid Kernel.	S1, S4
B-LibSVM ν-SVC Linear Kernel, B-LibSVM ν-SVC Poly Kernel, B-LibSVM ν-SVC Radial Kernel, B-LibSVM ν-SVC Sigmoid Kernel, LibSVM ν-SVC Linear Kernel, LibSVM ν-SVC Polynomial Kernel, LibSVM ν-SVC Radial Kernel, LibSVM ν-SVC Sigmoid Kernel.	S4

which are SVM, J48, Naive Bayes, Random Forest, SMO, and JRIP.

SVM was the most commonly used, being applied in 6 of the 17 studies (35.29%) and the second most commonly used machine learning techniques are J48, Naive Bayes, Bayesian Belief Networks, and Random Forest, being applied in 4 of the 17 studies (30.77%). Table 9 shows the detailed information.

We can observe that SVM, J48, Naive Bayes, Random Forest, SMO, JRIP received the most attention in 2013, 2016, and 2018 while in 2017, only Neural Networks received attention. We further observe that the authors of the primary studies applied the most commonly used machine learning techniques.

In general, the machine learning technique is categorized into two types: supervised and unsupervised. Figure 5 shows this taxonomy [95, 96]. According to our findings, we observe that all the applied techniques in the field of code smell detection were supervised machine learning techniques. The findings were expected since the application of supervised techniques is the most often applied in studies [97, 98].

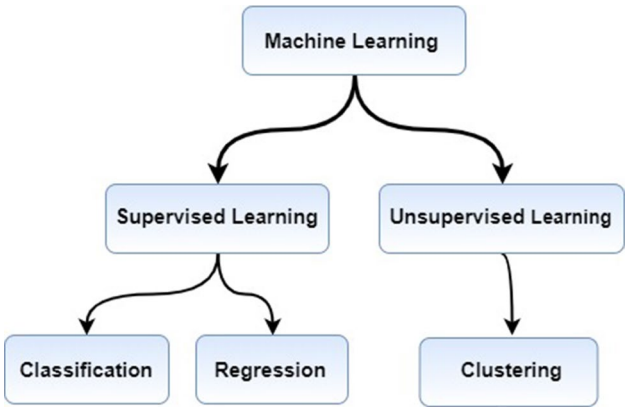

Fig. 5 Machine learning taxonomy [95, 96]

4.3 Detected Smells (RQ2)

Twenty-eight types of code smells were used in the PSs. Table 12 presents our findings. We found that 12 of the PSs detected more than one code smell, while the other five studies detected one code smell. The table lists the used code smells in descending order from the highest frequency to the lowest.

We found that the authors of studies **S1, S3, S4,** and **S7** not only used the same code smells, they also used the same number of code smells. The authors of **S4** and **S7** are the same, whereas the authors of study **S1** replicated study **S4**. **S5** used the highest number of code smells, using 11 code smells.

We observe that the God Class was the most used bad smell, being used in 13 studies (76.47%). The second most used bad smell was the Long Method which was used in nine studies (52.94%). The Feature Envy code smell was used in seven studies (41.17%), and the Data Class was used in five studies (29.41%). The remaining code smells were used in a smaller number of studies ranging from one to three.

4.4 The Accuracy of Machine Learning Techniques (RQ3)

To answer this question, we explore the evaluation metrics used in the PSs to evaluate the performance of the machine learning techniques used in code smell detection. Then, we compare the machine learning techniques that were used in code smell detection to identify the techniques that achieved the highest detection performance.

Several evaluation metrics were used by the PSs to evaluate the performance of the machine learning techniques used in code smell detection. These evaluation metrics are presented in Table 13 in descending order according to their frequency in the PSs. We notice that the recall metric was the most frequently used metric, being used in 9 of the 17

Table 12 Distribution of detected code smells over studies

Detected smells/studies	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15	S16	S17	Total	Frequency %
God class	√	√	√	√	√	√	√	√	√			√	√			√	√	13	76.47
Long Method	√		√	√	√	√	√			√	√		√					9	52.94
Feature Envy	√	√	√	√	√		√			√								7	41.17
Data Class	√	√	√	√			√											5	29.41
Lazy Class		√				√				√								3	17.64
Spaghetti Code Instances								√	√							√		3	17.64
Functional Decomposition (FD)								√	√							√		3	17.64
Large Class		√				√												2	11.76
Message Chains						√				√								2	11.76
Long Parameter list						√				√								2	11.76
Swiss Army Knife (SAK)						√		√										2	11.76
Duplicated Code														√	√			2	11.76
Parallel Inheritance Hierarchies		√																1	5.88
Type Checking					√													1	5.88
AntiSingleton						√												1	5.88
Refused Parent Request						√												1	5.88
Speculative Generality						√												1	5.88
Private Class Data						√												1	5.88
Middle Man										√								1	5.88
Complex Class						√												1	5.88
Empty catch block					√													1	5.88
Careless cleanup					√													1	5.88
Dummy handler					√													1	5.88
Unprotected main					√													1	5.88
Nested try statement					√													1	5.88
Exception thrown in finally block					√													1	5.88
Switch Statement										√								1	5.88
Over logging					√													1	5.88
# Detected smells	4	6	4	4	11	12	4	4	3	7	1	1	2	1	1	3	1		

Table 13 Performance evaluation methods

Validation Method	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15	S16	S17	Frequency
Recall			√			√		√	√	√		√			√	√	√	9
Accuracy	√	√		√			√			√	√		√	√				8
Precision			√			√		√	√			√				√	√	7
F-measure	√			√		√	√											4
ROC Area	√			√			√											3
Specificity										√								1
Mean absolute error (MAE)					√													1
Root mean square error (RMSE)					√													1

studies, followed by the accuracy metric, which was used in 8 of the 17 studies. Different configurations were used in the PSs to validate the learning mechanism. We present these mechanisms in Table 14. We observe that, 7 studies (**S1, S4, S5, S7, S10, S15** and **S16**) adapted the K-fold cross-validation method. This method guarantees that each fold includes the same proportions of the smelly and non-smelly classes. Our findings show that only 2 studies (**S1** and **S4**) used the grid-search algorithm.

The strategy which was followed to answer this question is presented in Tables 15 and 16. The main attributes of our evaluation are:

- The machine learning algorithm used
- The considered code smells
- The study source that mentions the evaluation
- A lower or a higher prediction (the sign of "+" means higher, while "−" is lower)

For example, in **S4** [7], the J48 algorithm performs better than naive Bayes to detect the code smell of Data Class, Long method, and Feature Envy. However, this study found that the naive Bayes algorithm is better than the J48 algorithm in terms of predicting the God Class code smell. Therefore, the algorithm in the row of the table is better than the algorithm in the column if the sign is "+" and the opposite in case the sign is "−."

For more clarification, Table 17 compares the used machine learning techniques per the considered code smell. The best machine learning algorithm for detecting the Data Class smell is the J48 algorithm, followed by JRip. For the Long Method smell, the JRip algorithm is the best followed by J48. Random Forest is the best at detecting the Feature Envy code smell, whereas the Naive Bayes algorithm is the best at detecting God Class. In conclusion, the standalone machine learning algorithm that consistently exceeds the other standalone machine learning algorithms was the J48 algorithm, whereas the worst algorithm was VFI. Table 17 sorts these algorithms for each code smell based on the reported accuracy.

The remainder of the studies applied one standalone machine learning technique to detect the type of code smell. Table 18 depicts the evaluation conducted by these studies. In study **S3** [70], the authors applied SVM to detect God Class, Data Class, Long Method, and Feature Envy. The results of this study show that the SVM achieved the highest accuracy in detecting the God Class code smell. Additionally, **S8** [73] applied SVM to detect four types of code smells namely: Spaghetti Code, Functional Decomposition, Swiss Army Knife. Their results support **S3** [70] in the case of detecting God Class. Consequently, the SVM algorithm has a good estimation of the God Class code smell. In **S13** [76], the Decision Tree Forest is more accurate for detecting God Class than the Long Method.

Some studies used ensemble techniques. Therefore, we need to compare the performance of these techniques to find the machine learning technique that performs better than the others. We follow the same strategy to compare the standalone machine learning techniques. Table 19 shows the studies which used the boosting techniques.

Three of the 17 selected studies used boosting techniques (**S1** [68], **S4** [7], **S7** [72]). They applied the same algorithm and detected the same types of code smell, and there is also a high similarity in the datasets used, especially **S1** [68], **S4** [7]. The authors in study **S1** [68] replicated the experiment conducted in study **S4** [7] and made some modifications to the datasets that were built in **S4** [7].

Table 20 shows which strategy was applied in each of the studies. The main attributes of our evaluation are the name of the machine learning algorithm used, the detected code smell, the study source that mentioned the evaluation, and a

Table 14 Validation of the learning mechanism

Validation mechanism	Studies
k-fold cross-validation	S1, S4, S5, S7, S10, S15, S16
Leave-one-out method	S13

Table 15 Standalone machine learning accuracy comparison

	Naive Bayes		Random Forest		JRIP		SVM		SMO	
	+	−	+	−	+	−	+	−	+	−
J48	(S1, S4) DC, FE, (S4) LM	(S1, S4) GC, (S1) LM	(S1, S4) GC, (S1) DC, GC, FE, (S1) LM	(S4) DC, LM, FE	(S1, S4) DC, GC, FE, (S1) LM	(S4) LM, FE	(S1, S4) DC, GC, FE, LM		(S1, S4) DC, GC, FE, LM	
Naive Bayes			(S1), DC, LM, FE, (S1, S4) GC	(S4), DC, LM, FE	(S1, S4) GC, (S1) LM, FE	(S1, S4) DC, (S4) LM, FE	(S1, S4) DC, GC (S1) LM, FE	(S4) LM, FE	(S1, S4) GC, (S1) DC, LM, FE	(S4) DC, LM, FE
Random Forest					(S4) DC, LM, FE, GC	(S1) DC, LM, FE, GC	(S4) DC, LM, FE, GC	(S1) DC, LM, FE, GC	(S4) DC, LM, FE, GC	(S1) DC, LM, FE, GC
JRIP							(S1, S4) DC, LM, FE, GC		(S1, S4) DC, LM, FE, GC	
SVM									(S1, S4) GC, (S1) DC, LM, FE	(S4) DC, LM, FE

GC God class, *DC* Data class, *LM* Long method, *FE* Feature envy, *SS* Switch statement, *LP* Long parameter list, *MC* Massage chains, *MM* Middle man, *LC* Lazy class

Table 16 Standalone machine learning accuracy comparison continued

	Naive Bayes		Random Forest		Binary logistic regression (BLR)		IBK		VFI		IBL	
	+	−	+	−	+	−	+	−	+	−	+	−
J48	(S1, S4) DC, (S10) FE, (S10) LM, (S10) LC, SS, LP, MM	(S10) MC	(S10) LM, LC, SS, MM	(S10) FE, LP, MC, SS, MM	(S10) LM, FE, LC, SS, MM	(S10) LP, MC	(S10) LC, SS, MM	(S10) LM, FE, LP, MC	(S10) LC, SS, MM, LM, FE, LP, MC		(S10) LC, SS, MM	(S10) LM, FE, LP, MC
Naive Bayes				(S10) LM, FE, LC, SS, MM, LP, MC		(S10) LC, SS, MM, LM, FE, LP, MC		(S10) LC, SS, MM, LM, FE, LP, MC	(S10) LC, SS, MM, LM, FE, LP, MC		(S10) LC, SS, MM, LM, FE, LP, MC	
Random Forest					(S10) LC, SS, MM, LM, FE, LP	(S10), MC	(S10) LC, FE, LP	(S10) LM, SS, MM, MC	(S10) LC, SS, MM, LM, FE, LP, MC		(S10) SS, LC, FE, LP	(S10) LM MM, MC
BLR							(S10) FE	(S10) LM, LC, LP, SS, MM	(S10) FE, LM, LC, LP, SS, MM		(S10) FE, LM, LC, LP, SS, MM	
IBK									(S10) FE, LM, LC, LP, SS, MM		(S10) LM, LC, LP, SS, MM	(S10) FE
IBL									(S10) FE, LM, LC, LP, SS, MM			

Table 17 Suitable machine learning technique to detect a specific code smell

Code smell	Suitable machine learning technique
Data Class	J48, JRIP, Random Forest, Naive Bayes, SMO, SVM.
Long Method	J48, JRIP, IBK, Naive Bayes, Random Forest, IBL, Binary Logistic Regression, SMO, SVM, VFI.
Feature Envy	Random Forest, J48, IBK, Naive Bayes, JRIP, IBL, SMO, SVM, Binary Logistic Regression, VFI.
God Class	Naive Bayes, J48, Random Forest, JRIP, IBK, SMO, SVM, IBL, Binary Logistic Regression, VFI.
Long Parameter List	Random Forest, IBK, Binary Logistic Regression, J48, IBL, Naive Bayes, VFI.
Middleman	J48, IBK, IBL, Random Forest, Binary Logistic Regression, Naive Bayes, VFI.
Message Chains	IBL, IBK, Binary Logistic Regression, Random Forest, Naive Bayes, J48, VFI.
Lazy class	J48, Random Forest, IBK, Binary Logistic Regression, IBL, Naive Bayes, VFI.
Switch Statement	J48, IBK, Random Forest, Binary Logistic Regression, IBL, Naive Bayes, VFI.

Table 18 Standalone machine learning accuracy comparison

Model	Suitable for
SVM	(S3, God Class, Data Class, Long Method, Feature Envy), (S8, God Class, Spaghetti Code, Swiss Army Knife, Functional Decomposition)
Decision Tree Forest	(S13, God Class, Long Method)
Binary Logistic Regression	(S11, Long Method)
Bayesian Belief Networks	(S11, God Class), (S9, God Class, Swiss Army Knife, Functional Decomposition, Spaghetti Code), (S15, Duplicated Code), (S17, God Class)
Natural language processing	(S14, Duplicated Code)

Table 19 Application of boosting techniques

Techniques	Studies
Standalone machine learning	S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17
Boosting	S1, S4, S7

lower or a higher detection ("+" indicates higher while "−" indicates lower).

Table 21 presents the code smell and which machine learning algorithm has performed better than the others in relation to detecting the code smell. Table 21 shows that the best machine learning algorithm to detect Data Class smell is the B-J48 pruned algorithm. The second most effective algorithm is the B-JRip, followed by the B-Random Forest algorithm. The B-J48 unpruned algorithm was the best at detecting the Long Method smell, followed by B-JRip. The B-J48 algorithm and the B-JRip algorithm were the two most effective in detecting all code smells except for God Class where B-Random Forest was the second most effective algorithm. We conclude that the family of J48 techniques and B-Random Forest are the top techniques for all types of code smells, while the SVM techniques are the worst. Furthermore, we note that, in some cases, using the boosting technique on the models does not always enhance their performance, and in fact, it could make the model worse. For instance, in **S4** [7], using naive Bayes to predict the God Class code smell without using boosting techniques was the best algorithm but after applying boosting techniques, its performance declined. This was also the case when using Random Forest with boosting techniques to detect God Class.

We observe that the highest performance results in all PSs obtained by B-J48 unpruned as it achieved 99.63% of F-Measure value when it was used to detect Long method code smell, followed by B-J48 pruned which achieved 99.26% of F-Measure value when it was used to detect Data Class code smell.

4.5 The Used Datasets (RQ4)

In this section, we analyze the datasets that were used for code smell detection and identify several attributes of these datasets, such as the dataset name, size, type (commercial, student, open source), availability, and language.

Table 22 presents general information on the datasets used in the 17 studies. Fourteen studies used open-source systems, while three studies (**S10**, **S13**, and **S15**) used industrial systems. All studies used systems written in the Java except for **S14**, **S15**, they used systems written in C and C#, respectively. Most studies used software metrics as the independent variables, as shown in Table 23, except for S14 they used textual metrics called string tokenization. Two studies of the 17 made their dataset available. Table 22 presents the number of systems used to build their datasets, the dataset

Table 20 Comparison of accuracy of boosting techniques

	B-naive Bayes		B-random forest		B-JRIP		B-SVM		B-SMO	
	+	−	+	−	+	−	+	−	+	−
B-J48	(S1, S4, S7) DC, GC, (S4, S7) LM, FE	(S1) LM, FE	(S1, S4) DC, FE, GC, (S1) LM	(S7, S4) LM, (S7) DC, FE, GC	(S1, S4, S7) DC, GC, LM, FE		(S1, S4, S7) DC, GC, LM, FE		(S1, S4, S7) DC, GC, LM, FE	
B-Naive Bayes			(S1) DC, GC, LM, FE	(S4, S7) DC, GC, LM, FE	(S1) GC, FE	(S1, S4, S7) DC, LM, (S4, S7) GC, FE	(S1, S4, S7) DC, (S1, S4) LM, (S4) GC, (S1) FE.	(S7) LM, (S1, S7) GC (S4, S7) FE, (S1) DC.	(S1, S4) LM, FE, (S1) GC	(S1, S4, S7) DC (S4, S7) GC, (S7) LM, FM
B-Random Forest					(S4, S7) GC, (S4) LM, (S7) DC	(S1, S7) LM, (S1, S4, S7) FE, (S1, S4) DC, (S1) GC	(S4, S7) DC, LM, GC, FE	(S1) DC, LM, GC, FE	(S4, S7) DC, LM, GC, FE	(S1) DC, LM, GC, FE
B-JRIP							(S1, S4, S7) DC, LM, FE, (S4, S7) GC	(S1) GC	(S1, S4, S7) DC, LM, FE, (S4, S7) GC	(S1) GC
B-SVM									(S1) LM, FE, GC	(S1, S4, S7) DC, (S4, S7) GC, LM, FE

Table 21 Suitable machine learning algorithm to detect a specific Code smell

Code smell	Suitable machine learning algorithm
Data Class	B-J48, B-JRIP, B-Random Forest, B-SMO, B-SVM, B-Naive Bayes.
Long Method	B-J48, B-JRIP, B-Random Forest, B-Naive Bayes, B-SMO, B-SVM.
Feature Envy	B-J48, B-JRIP, B-Random Forest, B-Naive Bayes, B-SMO, B-SVM.
God Class	B-J48, B-JRIP, B-Random Forest, B-SMO, B-SVM, B-Naive Bayes.

Table 22 A summary of previous studies

Study	Dataset	# Systems	Dataset inputs	Dataset type	Language	#Used metrics
S1	–	74	Object-oriented metrics	Open source	Java	36
S2	–	20	Object-oriented metrics	Open source	Java	8
S3	–	2	Object-oriented metrics	Open source	Java	–
S4	Available	74	Object-oriented metrics	Open source	Java	82
S5	–	4	Object-oriented metrics	Open source	Java	17
S6	–	4	Object-oriented metrics	Open source	Java	62
S7	–	76	Object-oriented metrics	Open source	Java	82
S8	Available	3	Object-oriented metrics	Open source	Java	50
S9	–	2	Goal Question Metric (GQM)	Open source	Java	–
S10	–	-	Object-oriented metrics	Industrial	Java	27
S11	–	1	Expert's knowledge & Object-oriented metrics	Open source	Java	–
S12	–	2	Rule based	Open source	Java	–
S13	–	2	Object-oriented metrics	Industrial	Java	9
S14	–	4	Textual	Open source	C	–
S15	–	2	Object-oriented metrics	Industrial	C#	21
S16	–	2	Goal Question Metric (GQM)	Open source	Java	–
S17	–	2	Object-oriented metrics	Open source	Java	–

input type, dataset type, language, and the number of used metrics.

Table 24 shows the size of each dataset in terms of three attributes: the number of used systems, the number of classes, and the number of lines of code. We note that only three studies used more than 20 systems. Therefore, we can report these as reliable datasets. Some of the studies did not mention the size (number of classes and the number of lines of code) of their datasets such as (**S3**, **S6**) but they mentioned the number of systems. In **S3**, there are 20 systems, and in **S7**, there are 76 systems. The authors of **S10** did not mention the attributes (the number of used systems, the number of classes and the number lines of code) as they only collected 7 datasets from the previous literature. We observe that 14 PSs used open-source systems. Therefore, they are freely available and can be used to replicate the research. Furthermore, other researchers who utilized the same systems allow comparison. Additionally, previous studies reported that commercial systems may provide higher code quality than open-source systems [99].

Table 25 shows the four most used systems used in the 17 PSs, while Table 23 presents the software metrics used as independent variables in the datasets. We can see that only eight of the seventeen studies mentioned the software metrics. These metrics include the metrics proposed by Chidamber and Kemerer in 1991[1] [100, 101], the MOOD metrics proposed by Abreu and Carapua in 1994[2] [102–104], and the QMOOD metrics proposed by Bansiya and Davis in 2002[3] [105–107]. We observe that the C&K software metrics were used in five of the eight studies that mentioned software metrics as C&K metrics cover different object-oriented properties, while the QMOOD software metrics were used in two studies. The MOOD software metrics were used in one study. To obtain the values of these metrics from the systems, several tools were used. These tools are shown in Table 26. From Table 26, we observe that the tools POM and design features and metrics for Java (DFMC4 J) were used in three of the seventeen studies. The DFMC4J tool was used in two studies **S4** and **S7** which have the same authors, and it was also used in **S1** which replicated the experiment in **S4**.

[1] C&K metrics (DIT, NOC, CBO, RFC, WMC, LCOM).

[2] MOOD metrics (MHF, AHF, MIF, AIF, POF, COF).

[3] QMOOD metrics (DSC, NOH, ANA, DAM, DCC, CAM, MOA, MFA, NOP, NOM, CIS).

Table 23 Software metrics used in previous studies

Authors	Software metrics
Kim [69]	LOC, CC, C&K metrics
Fontana et al. [7]	C&K metrics except (LCOM), LOC, LOCNAMM, NOPK, NOCS, NOM, NOMNAMM, NOA, CYCLO, WMCNAMM, AMW, AMWNAMM, MAXNESTING, CLNAMM, NOP, NOAV, ATLD, NOLV, FANOUT, FANIN, ATFD, FDP, CFNAMM, CINT, MaMCL, MeMCL, NMCS, CC, CM, NOAM, NOPA (NOAP), LAA, NOI, NMO, NODM, NOPM, NOPRM, NOPLM, NONAM, NOSM, NIM, NOII, NODA, NOPVA, NOPRA, NOFA, NONCM, NOFM, NOFNSM, NOFSM, NONFNABM, NOFSA, NOFNSA, NONFNSA, NOSA, NONFSA, NONFNSA, NOSA, NONFSA, NOABM, NOCM, NONFNSM, NONFSM
Tarwani et al. [52]	LOC, C&K metrics, CC, Cyclomatic complexity (Ocavg), Dependencies (Dcy &Dcy), Javadoc function (jf), Javadoc Line of code (jLOC), Javadoc methods (jm), MHF, AHF
Amorim et al. [71]	C&K metrics, QMOOD metrics except (CAM, ANA, CIS), ACAIC, ACMIC, AID, AMC, CA, CAM, CE, CLD, COHESIONATTRIBUTES, CONNECTIVITY, DCAEC, DCMEC, DIT 1, IC, ICHCLASS, IR, LCOM1, LCOM2, LCOM3, LCOM5, LOC, LOC 1, MCCABE, MLOCSUM, NAD, NADEXTENDED, NCM, NMA, NMD, NMDEXTENDED, NMI, NMO, NOA, NOC 1, NOD, NOF, NOPARAM, NOPM, NOTC, NOTI, NPM, RFC NEW, SIX, VGSUM, WMC NEW, WMC1, CBM
Maiga et al. [73]	C&K metrics, QMOOD metrics, ACAIC, ACMIC, AID, CBOin, CBOout, CLD, DCAEC, DCMEC, ICHClass, IR, LCOM2, LCOM5, LOC, McCabe, NAD, NADExtended, NCM, NMA, NMD, NMDExtended, NMI, NMO, NOA, NOD, NOPM, NOParam, NOTI, NPrM, SIX, USELESS, WMC1, cohesionAttributes, connectivity
Maneerat et al. [32]	MOOD metrics, NA, NC, NM, NO, NP, C PARAM, RFC, WAC, WMA, D APPEAR, DIT, NOC, NAI, NOI, ACT, COMP, NS, CBC, ABSTR R, ASSOC R, DEPEND R
Kreimer [76]	Number of statements of a method, NOLV, NOP, CYCLO, Number of instance variables of a class, Median of the number of statements of all methods of a class, Number of internal connected components, Median of complexities of all methods of a class, Number of external connected components
Wang et al. [78]	"Whether it is a Local Clone, Method Name Similarity, Masked File Name Similarity, Sum of Parameter Similarities, Difference on Only Post x Number, Maximal Parameter Similarity, Fine Name Similarity, LOC, Number of Library Invocations Number of Other Invocations, Number of Local Invocations, Number of Invocations, Number of Library Invocations, Number of Invocations, Number of Other Invocations, Number of Field Accesses, Number of Local Invocations, Number of Recent Changes, Number of Changes, Existence Time, File Existence Time, Number of Recent File Changes, Number of File Changes"

Table 24 Dataset size

#Systems	#Classes	#LOC	Study
74	51,826	6,785,568	S1, S4
2		5,594,000	S15
20	53,000	3,000,000	S2
4	13,700	4,070,000	S6
4	1089		S5
3	3192	113,017	S8
2	777	271,000	S9, S12, S16
2	688	96,889	S13
1	193	20	S11

Regarding the dependent variables in each dataset, 12 studies used binary classification ("0" or "1") or ("smelly" or "non-smelly"), while only two studies **S1** and **S16** used multiclassification.

4.6 The Tools Used to Implement the Machine leArning Algorithms (RQ5)

In the PS, we found that two kinds of tools were used to implement the machine learning algorithms to detect code smells: WEKA and Tensor Flow implemented in Python. Table 27 shows these tools which were used in the **PSs** to

Table 25 Used systems

System	S1	S2	S3	S4	S6	S7	S8	S9	S12	S13	S14	S15	S16	S17	Frequency
Xercesv	√		√	√		√	√	√	√				√	√	9
ArgoUML	√		√	√	√	√	√						√		7
WEKA	√			√		√				√					4
junit-4.10	√	√		√		√									4

Table 26 Tools used to compute the metrics

Tool	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13	Frequency
POM						√		√				√		3
Design Features and Metrics for Java (DFMC4J)	√			√			√							3
Eclipse Plugin			√								√			2
SciTools Understand		√												1
Analyst 4J			√											1
IntelliJ IDEA					√									1
CKJM						√								1
Design Program Dependence Graph (DPDG)													√	1

Table 27 Tools used to implement experiments

Tool	Studies	#Studies
Weka	S1, S3, S4, S6, S7, S8, S9, S10, S12, S13, S15	11
Tensor flow	S2	1
Not mentioned	S5, S11, S14, S16, S17	5

implement machine learning algorithms. Eleven of the 17 studies used Weka, and only one study **S2** utilized Tensor Flow written in Python. Five studies **S5, S11, S14, S16,** and **S17** did not provide details on the tool that was used.

5 Discussion

In this section, we discuss the main results of this study based on the research questions. Furthermore, we report some recommendations that can be used as a start point for the future research.

5.1 Types of Machine Learning Techniques Used (RQ1)

Our findings show that 28 of the most commonly used machine learning techniques were used to detect code smells. The reasons behind using these techniques in many studies are: the high performance these techniques provide and the availability of these techniques in tools. Only 5 out of 17 **PSs** used more than one machine learning algorithms; this reveals a lack in the application of machine learning to address the issue of smells in the software. Therefore, more studies are needed in this field. Future studies may need to use more than one machine learning technique to explore the capability of each technique to detect code smells on the same dataset.

Ensemble learning is proposed to improve the performance of weak classifiers by combining several single classifiers using different methods. However, most of the reviewed studies applied a single classifier whereas three studies out of the seventeen used boosting-based ensemble techniques (homogenous ensemble techniques). Ensemble techniques were found to outperform single learning techniques in predicting software defects [108]; thus, more studies that use ensemble techniques to detect code smells should be conducted.

5.2 Detected Smells (RQ2)

Twenty-eight types of code smells were detected in the PSs. God Class, Long Method, Feature Envy, and Data Class were the most detected smells; they also got more attention in the recent studies, as shown in Table 12. The possible reasons behind selecting these code smells are:

- They cover different design problems.
- Their critical influence on software quality.
- They are the most frequently occurring code smells.
- These code smells are well known and easy to understand.
- The availability of these code smells in the automated tools.

In terms of using machine learning to detect code smells, we observe that most of the PSs used a small number of code smells. Furthermore, not all code smells proposed by Fowler [33] are covered by the previous studies so some of these code smells that might have an effect on the software quality have not been studied yet; hence, more studies can be conducted to study these code smells.

5.3 The Accuracy of Machine Learning Techniques (RQ3)

We observe that the recall metric was the most frequently used metric followed by the accuracy and precision metrics. However, some researchers (for instance the authors of [7]) claim that the metrics accuracy, F-measure, and ROC Area show several points of views of the performances of predictive models and showed that using only one of these measures can show the whole performance measure; therefore, these metrics should also be used to evaluate the performance of applied machine learning techniques for code smells detection; however, recall and precision should be used together to adequately assess the performance of a model. In addition, they reported that it is not recommended to use accuracy metric alone especially in the case of having unbalanced datasets (negative and positive classes are unbalanced). We observed that most of the used performance metrics are important measures for the classification models: recall, accuracy, precision, F1-score, AUC, and specificity [109–111]. Only two metrics used for a regression problem which are RMSE and MAE measures. These two metrics were proposed to use for regression models [112, 113]. Furthermore, only two studies applied the grid-search algorithm. This algorithm used to search and find the optimal parameters of machine learning algorithms that yield the highest performance. Therefore, we believe that more studies are needed to address this issue to generate more efficient code smell detection models.

5.4 The Used Datasets (RQ4)

Based on our findings, the Java language got more attention from the previous studies. Most of the PSs tested their approaches using systems written in java. For this reason, more studies are needed to investigate the capabilities of machine learning techniques to detect code smells in other programming languages such as C#, C ++. Moreover, we found that, five studies (**S1, S2, S4, S7** and **S6**) build their datasets from different systems, while the other studies build their datasets based on one system. Building datasets from different systems is the best way to avoid the issue of bias. In addition, most of PSs used software metrics as the independent variables, while recent studies [114, 115] have reported that software metrics are insufficient to optimize software quality. To address this issue, the implementation of other resources should be used for identifying the characteristic of smelly code such as dynamic, historical, and textual aspects. In addition, the manual validation process of the dataset is an important step to have an effective dataset.

Feature selection is an essential preprocessing step in machine learning that aims to select a small subset of the relevant features from the original dataset by eliminating redundant, irrelevant or noisy features in order to achieve better learning performance, i.e., enhances accuracy, lower computational cost and time [116]. Our findings show that only one study **S1** used Gain Ratio Feature Evaluation technique as feature selection techniques. Thus, more studies are needed to investigate the use feature selection in code smells detection. On the other hand, only two studies applied multiclassification. Therefore, future studies should consider multiclassification.

Table 28 Bad smell detection tools

Detectors	Code Smell
iPlasma	Long Method, Data Class, Feature Envy, God Class
Marinescu	Long Method
Fluid Tool	Feature Envy, Data Class
Antipattern Scanner	Data Class
PMD	Long Method, God Class

5.4.1 Dataset Analysis

In this subsection, we analyze the available datasets that were used in **S4** [7] and **S8** [73]. The first dataset was built by Arcelli Fontana et al. [7] who collected systems from one of the massive curated benchmark datasets, called Qualitas Corpus [117], release 20120401r. 74 systems out of 111 Java systems of the corpus were selected. Thirty-seven systems were ignored since they could not be compiled; consequently, bad smell detection could not be achieved. In these 74 software systems, there are 6,785,568 lines of code, 3420 packages, 51,826 classes, and 404,316 methods.

The software metrics presented in Table 23 were used as independent variables in their machine learning approaches for detecting class and method level smells. At the class level, there were two smells: Data Class and God Class and at the method level, there were also two smells: Feature Envy and Long Method. At the method level, 82 software metrics were computed for each system, while at the class level, only 61 software metrics were computed. The computed operation was done using a tool developed by the authors called "design features and metrics for Java (DFMC4J)."

Four datasets were built by the authors of **S5,** one for every smell, i.e., every dataset included code that has smelly and non-smelly parts. Table 28 shows the set of automatic detectors that were employed by the authors to detect each code smell to establish the dependent variable for the bad smell detection methods.

Generally, achieving 100% recall is unusual in this field, meaning that an automatic detection method might not recognize real bad smell instances even when combining

Table 29 Sampling procedure

Code case	Instances
Smelly	826
Non-smelly	1160
Total	1986

multiple detectors. A stratified random sampling of the methods/classes of the studied systems was utilized to enhance their confidence in validity of the dependent variable and address the false positives: this sampling performed 1,986 instances as shown in Table 29, the balance of the training set being 2/3 negative and 1/3 positive instances, where positive means the system is affected by a code smell, while negative means it is not affected. Later, a manual validation of these instances was conducted by three master students to check the detection results.

The authors made a normalization of the sampled dataset: by removing the non-smelly and smelly elements and building four separate datasets using every kind of software smell, containing 280 non-smelly and 140 smelly instances; hence, there were 420 instances in each dataset. The training set for the machine learning methods was represented by these four datasets.

The second dataset was built by Maiga et al. [73, 118] who analyzed three open-source systems that are freely available and have been utilized in several studies: Azureus v2.3.0.6, Xerces v2.7.0, and ArgoUML v0.19.8.

The software metrics presented in Table 23 were used as independent variables in their machine learning technique for detecting four code smells: God Class, Swiss Army Knife (SAK), Functional Decomposition (FD), and Spaghetti Code (SC). These metrics were used as training inputs. They calculated the values of the software metrics to use them as independent variables, and the computed operation was done using a tool called POM. This tool is an extensible framework that depends on the PADL metamodel [119] and contains more than 60 software metrics [120].

Maiga et al. built three datasets for every system and every code smell. These datasets comprise two parts: smelly or non-smelly classes in equal numbers. They used 30 functional decomposition classes in Azureus v2.3.0.6 to build each dataset and added 30 non-functional decomposition classes by selecting them randomly in the remaining classes of Azureus v2.3.0.6. After this, they divided the 60 classes randomly into the three datasets: Dataset1, Dataset2, and Dataset3, making sure that each dataset contains 10 functional decomposition instances and 10 functional decomposition instances. Therefore, every dataset has 20 instances. Finally, they used practitioners' feedback on the results as input for the training dataset.

5.5 The Tools Used to Implement the Machine Learning Algorithms (RQ5)

We observe that most of the studies used Weka because this tool is the most common and widely used for implementing machine learning algorithms [7, 32]. In addition, this tool contains 37 algorithms for classification problems [32] and is considered a reliable tool [68]. Furthermore, it is freely available. However, based on our findings, we can claim that using only two tools in this field is not enough since the used tools do not provide all types of machine learning techniques. This is a clear opportunity for many researchers to improve the detection of code smells by using other application environments which are considered as in the top of ranking for implementing machine learning techniques and also have numerous libraries which are the implementations of recent machine learning techniques, for example, Python, R, and Julia.

6 Threats to Validity

In this section, we identify some potential threats to validity which might have biased our systematic literature review. The potential threats to the validity of our research are searching and choosing the relevant studies and the data extraction processes.

The authors in [31, 121] reported that a common threat to SLR is the process of finding all the relevant research. To find the relevant research, one threat is the process of selecting online databases. To mitigate this threat, five online research databases were selected. We selected the well-known online databases that cover all journal papers and conferences: ACM, IEEE, Scopus, Springer, and Science Direct.

In order to mitigate the threat of excluding a relevant study or including an irrelevant study in the study selection phase, three researchers undertook the selection process and resolved all conflicts in the case of disagreement.

To mitigate bias in the data extraction phase, data extraction cards were constructed. Three researchers performed this process, one of them extracting the data then inserting this into the cards, while the others examined the collected data. In the case of disagreement, a discussion was conducted to resolve these conflicts.

7 Conclusion

In this study, we performed an SLR for empirical studies on the machine learning techniques used to detect code smells. The main objective was to systematically review and analyze the machine learning techniques applied to detect code

smells from different perspectives: the code smells that were employed in the experiments, the types of machine learning techniques applied to detect code smells, a comparison between these models in terms of prediction accuracy and performance, the datasets used, and the tool utilized to implement the machine learning models.

A literature search was conducted using five online databases to retrieve the relevant studies. As a result, seventeen primary studies relating to the five research questions in this study were selected. The results show that God Class and Long Method, Feature Envy, and Data Class are the most frequently detected code smells. We also observed that machine learning algorithms SVM, J48, Naive Bayes, Random Forest, BBN, SMO, and JRIP are used the most. We also observed that the standalone machine learning algorithms that consistently outperform the other standalone machine learning algorithms are the J48 algorithm, Random Forest, JRip, Naive Bayes, IBk, IBL, SVM, SMO, and BLR. The algorithm with the worst performance is VFI.

Our review also indicated that the family of J48 techniques and B-Random Forest are the top techniques for detecting all types of code smells, while the SVM techniques are the worst. As well, we noticed that, in some cases, the use of boosting techniques on the models does not always enhance their performance. With respect to the dataset, it was found that most of the PSs used Java-based case studies and most of them used software metrics as independent variables. Only two of the 17 studies made the dataset available. Finally, we found that Weka is the most commonly used machine learning tool in the PSs, while one study employed Tensor Flow in Python. Even though R is popular for implementing the ML algorithms with excellent library support, none of the primary studies used it; hence, R could be used in future studies.

As a result of this study, we make the following observations to be considered when conducting new research that uses machine learning for bad smell detection: (1) B-J48 unpruned achieved the highest performance results in all PSs, it got 99.63% of F-Measure value when it was applied for Long method detection, followed by B-J48 pruned which achieved 99.26% of F-Measure value when it was applied for Data Class detection, (2) some code smells have not been investigate, (3) the use of ensemble learning techniques for code smells detection has gained less attention as it was used in two studies only, (4) only two studies used the grid-search algorithm to configure the machine learning algorithms parameters to find optimal parameters, (5) five studies built their datasets from different systems, (6) only two studies used multiclassification, and (7) only one study used feature selection technique.

It can be seen that the application of machine learning techniques to detect code smells is still a new area and needs further investigation. Therefore, more research effort should be focused to facilitate the employment of machine learning techniques to address the issue of predicting code smells.

Acknowledgement The authors acknowledge the support of King Fahd University of Petroleum and Minerals in the development of this work.

References

1. Guzel, A.; Aktas, Ö.: A survey on bad smells in codes and usage of algorithm analysis. Int J Comput Sci Softw Eng (IJCSSE) **5**(6), 114 (2016)
2. Danphitsanuphan, P.; Suwantada, T.: Code smell detecting tool and code smell-structure bug relationship. In: *2012 Spring Congress on Engineering and Technology,* pp. 1–5
3. Santos, J.A.M.; Rocha-Junior, J.B.; Prates, L.C.L.; Do Nascimento, R.S.; Freitas, M.F.; De Mendonça, M.G.: A systematic review on the code smell effect. J. Syst. Softw. **144**, 450–477 (2018)
4. Kruchten, P.; Nord, R.L.; Ozkaya, I.J.I.S.: Technical debt: from metaphor to theory and practice. IEEE Softw. **29**(6), 18–21 (2012)
5. Seaman, C.; Guo, Y.: Measuring and Monitoring Technical Debt, vol. 82, pp. 25–46. Advances in ComputersElsevier, Amsterdam (2011)
6. Khomh, F.; Vaucher, S.; Guéhéneuc, Y.G.; Sahraoui, H.: A bayesian approach for the detection of code and design smells. In: 2009 Ninth International Conference on Quality Software, pp. 305–314
7. Fontana, F.A.; Mäntylä, M.V.; Zanoni, M.; Marino, A.: Comparing and experimenting machine learning techniques for code smell detection. Empir. Softw. Eng. **21**(3), 1143–1191 (2016)
8. Sjøberg, D.I.; Yamashita, A.; Anda, B.C.; Mockus, A.; Dybå, T.: Quantifying the effect of code smells on maintenance effort. IEEE Trans. Softw. Eng. **39**(8), 1144–1156 (2012)
9. Yamashita, A.; Moonen, L.; (2012) Do code smells reflect important maintainability aspects?. In: 2012 28th IEEE international conference on software maintenance (ICSM), IEEE, pp. 306–315
10. Banker, R.D.; Datar, S.M.; Kemerer, C.F.; Zweig, D.: Software complexity and maintenance costs. Commun. ACM **36**(11), 81–95 (1993)
11. Roy, R.; Stark, R.; Tracht, K.; Takata, S.; Mori, M.: Continuous maintenance and the future–foundations and technological challenges. CIRP Ann. **65**(2), 667–688 (2016)
12. Bavota, G.; Oliveto, R.; Gethers, M.; Poshyvanyk, D.; De Lucia, A.: Methodbook: recommending move method refactorings via relational topic models. IEEE Transactions on Software Engineering **40**(7), 671–694 (2013)
13. Marinescu, R.: Detection strategies: Metrics-based rules for detecting design flaws. In: *20th IEEE International Conference on Software Maintenance, 2004. Proceedings*, IEEE, pp. 350–359
14. Moha, N.; Gueheneuc, Y.-G.; Duchien, L.; Le Meur, A.-F.: Decor: a method for the specification and detection of code and design smells. IEEE Trans. Softw. Eng. **36**(1), 20–36 (2010)
15. Palomba, F.; Bavota, G.; Di Penta, M.; Oliveto, R.; Poshyvanyk, D.; De Lucia, A.: Mining version histories for detecting code smells. IEEE Trans. Softw. Eng. **41**(5), 462–489 (2015)
16. Palomba, F.; Tamburri, D.A.A.; Fontana, F.A.; Oliveto, R.; Zaidman, A.; Serebrenik, A.: Beyond technical aspects: How do community smells influence the intensity of code smells? IEEE Trans. Softw. Eng. **99**, 1 (2018)
17. Palomba, F.; Zaidman, A.; De Lucia, A.: Automatic test smell detection using information retrieval techniques. In: *2018 IEEE*

International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp. 311–322
18. Tsantalis, N.; Chatzigeorgiou, A.: Identification of move method refactoring opportunities. IEEE Trans. Softw. Eng. **35**(3), 347–367 (2009)
19. Munro, M.J.: Product metrics for automatic identification of" bad smell" design problems in java source-code. In: 11th IEEE International Software Metrics Symposium (METRICS'05), IEEE, pp. 15
20. Morales, R.; Soh, Z.; Khomh, F.; Antoniol, G.; Chicano, F.: On the use of developers' context for automatic refactoring of software anti-patterns. J. Syst. Softw. **128**, 236–251 (2017)
21. Zhang, M.; Hall, T.; Baddoo, N.: Code bad smells: a review of current knowledge. J. Softw. Evol. Process **23**(3), 179–202 (2011)
22. Singh, S.; Kaur, S.: A systematic literature review: Refactoring for disclosing code smells in object oriented software. Ain Shams Eng. J. **9**(4), 2129–2151 (2017)
23. Sharma, T.; Spinellis, D.: A survey on software smells. J. Syst. Softw. **138**, 158–173 (2018)
24. Mariani, T.; Vergilio, S.R.: A systematic review on search-based refactoring. Inf. Softw. Technol. **83**, 14–34 (2017)
25. Rasool, G.; Arshad, Z.: A review of code smell mining techniques. J. Softw. Evol. Process **27**(11), 867–895 (2015)
26. Fernandes, E.; Oliveira, J.; Vale, G.; Paiva, T.; Figueiredo, E.: A review-based comparative study of bad smell detection tools. In: Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering, pp. 18–18
27. Fontana, F.A.; Braione, P.; Zanoni, M.: Automatic detection of bad smells in code: An experimental assessment. J. Object Technol. **11**(2), 1–5 (2012)
28. Garcia, J.; Popescu, D.; Edwards, G.; Medvidovic, N.: Identifying architectural bad smells. In: 2009 13th European Conference on Software Maintenance and Reengineering, IEEE, pp. 255–258
29. Azeem, M.I.; Palomba, F.; Shi, L.; Wang, Q.: Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. Information and Software Technology **108**, 115–138 (2019)
30. Caram, F.L.; Rodrigues, B.R.; Campanelli, A.S.; Parreiras, F.S.: Machine learning techniques for code smells detection: a systematic mapping study. J. Softw. Eng. Knowl. Eng. **29**(02), 285–316 (2019)
31. Kitchenham, B.; Charters, S.: Guidelines for performing Systematic Literature Reviews in Software Engineering. Technical Report EBSE 2007-001, Keele University and Durham University Joint Report
32. Maneerat, N.; Muenchaisri, P.: Bad-smell prediction from software design model using machine learning techniques. In: 2011 Eighth International Joint Conference on Computer Science and Software Engineering (JCSSE), pp. 331–336 (2011)
33. Beck, K.; Fowler, M.; Beck, G: Bad smells in code: Refactoring: Improving the design of existing code, pp. 75–88 (1999)
34. Lehman, M.M.: Programs, life cycles, and laws of software evolution. Proc. IEEE Spec. Issue Softw. Eng. **68**(9), 1060–1076 (1980)
35. Tufano, M et al.: When and why your code starts to smell bad. In: Proceedings of the 37th International Conference on Software Engineering, vol. 1, pp. 403–414 (2015)
36. Kessentini, W.; Kessentini, M.; Sahraoui, H.; Bechikh, S.; Ouni, A.: A cooperative parallel search-based software engineering approach for code-smells detection. IEEE Trans. Softw. Eng. **40**(9), 841–861 (2014)
37. Marticorena, R.; López, C.; Crespo, Y.: Extending a taxonomy of bad code smells with metrics. In: Proceedings of 7th International Workshop on Object-Oriented Reengineering (WOOR), p. 6 (2006)
38. Sourcemaking, Bad code smells
39. Sourcemaking Refactoring (2018). https://sourcemaking.com/refactoring
40. Samuel, A.L.: Some studies in machine learning using the game of checkers. IBM J. Res. Dev. **3**(3), 210–229 (1959)
41. Ruck, D.W.; Rogers, S.K.; Kabrisky, M.; Oxley, M.E.; Suter, B.W.: The multilayer perceptron as an approximation to a Bayes optimal discriminant function. IEEE Trans. Neural Netw. **1**(4), 296–298 (1990)
42. Rosenblatt, F.: Principles of neurodynamics. perceptrons and the theory of brain mechanisms, 1961
43. Cortes, C.; Vapnik, V.: Support-vector networks. Mach. Learn. **20**(3), 273–297 (1995)
44. Aljamaan, H.I.; Elish, M.O.: An empirical study of bagging and boosting ensembles for identifying faulty classes in object-oriented software. In: 2009 IEEE Symposium on Computational Intelligence and Data Mining, pp. 187–194 (2009)
45. Vladimir, N.V.: The Nature of Statistical Learning Theory. Springer, Berlin (1995)
46. Broomhead, D.S.; Lowe, D.: Radial basis functions, multi-variable functional interpolation and adaptive networks, 1988
47. Friedman, N.; Geiger, D.; Goldszmidt, M.: Bayesian network classifiers. Mach. Learn. **29**(2–3), 131–163 (1997)
48. Castillo, E.; Gutierrez, J.M.; Hadi, A.S.: Expert Systems and Probabiistic Network Models. Springer, Berlin (1996)
49. Rish, I et al.: An empirical study of the naive Bayes classifier. In: IJCAI 2001 Workshop On Empirical Methods In Artificial Intelligence, vol. 3, pp. 41–46 (2001)
50. Breiman, L.: Random forests. Mach. Learn. **45**(1), 5–32 (2001)
51. Zhang, Y.; Haghani, A.: A gradient boosting method to improve travel time prediction. Transp. Res. Part C Emerg. Technol. **58**, 308–324 (2015)
52. Tarwani, S.; Chug, A.: Predicting maintainability of open source software using Gene Expression Programming and bad smells. In: 2016 5th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO), pp. 452–459 (2016)
53. Seber, G.A.; Lee, A.J.: Linear Regression Analysis. Wiley, Hoboken (2012)
54. Weisberg, S.: Applied Linear Regression. Wiley, Hoboken (2005)
55. Stork, E.D.H.; Duda, R.O.; Hart, P.E.; Stork, D.G.: Pattern Classification. Academic Internet Publishers, New York (2006)
56. Yuan, Y.; Shaw, M.J.: Induction of fuzzy decision trees. Fuzzy Sets Syst. **69**(2), 125–139 (1995)
57. Quinlan, J.R.: C4. 5: programming for machine learning. Morgan Kauffmann **38**, 48 (1993)
58. Vilela, J.; Castro, J.; Martins, L.E.G.; Gorschek, T.: Integration between requirements engineering and safety analysis: a systematic literature review. J. Syst. Softw. **125**, 68–92 (2017)
59. Gasparic, M.; Janes, A.: What recommendation systems for software engineering recommend: A systematic literature review. J. Syst. Softw. **113**, 101–113 (2016)
60. Tarhan, A.; Giray, G.: On the use of ontologies in software process assessment: a systematic literature review. In: Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering, ACM, pp. 2–11 (2017)
61. Khan, S.U.; Azeem, M.I.: Intercultural challenges in offshore software development outsourcing relationships: an exploratory study using a systematic literature review. IET Softw. **8**(4), 161–173 (2014)
62. Niazi, M.: Do systematic literature reviews outperform informal literature reviews in the software engineering domain? An initial case study. Arab. J. Sci. Eng. **40**(3), 845–855 (2015)

63. Muccini, H.; Sharaf, M.; Weyns, D.: Self-adaptation for cyber-physical systems: a systematic literature review. In Proceedings of the 11th international symposium on software engineering for adaptive and self-managing systems, ACM, pp. 75–81 (2016)
64. Mohammed, N.M.; Niazi, M.; Alshayeb, M.; Mahmood, S.: Exploring software security approaches in software development lifecycle: a systematic mapping study. Comput. Stand. Interfaces **50**, 107–115 (2017)
65. Mufti, Y.; Niazi, M.; Alshayeb, M.; Mahmood, S.: A readiness model for security requirements engineering. IEEE Access **6**, 28611–28631 (2018)
66. Dieste, O.; Padua, A.G.: Developing search strategies for detecting relevant experiments for systematic reviews. In First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007), IEEE, pp. 215–224 (2007)
67. Wen, J.; Li, S.; Lin, Z.; Hu, Y.; Huang, C.: Systematic literature review of machine learning based software development effort estimation models. Inf. Softw. Technol. **54**(1), 41–59 (2012)
68. Di Nucci, D.; Palomba, F.; Tamburri, D.A.; Serebrenik, A.; De Lucia, A.: Detecting code smells using machine learning techniques: are we there yet?. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 612–621 (2018)
69. Kim, D.K.: Finding bad code smells with neural network models. Int. J. Electr. Comput. Eng. (IJECE) **7**(6), 3613–3621 (2017)
70. Kaur, K.; Jain, S.: Evaluation of machine learning approaches for change-proneness prediction using code smells. In: Proceedings of the 5th International Conference on Frontiers in Intelligent Computing: Theory and Applications, pp. 561–572 (2017)
71. Amorim, L.; Costa, E.; Antunes, N.; Fonseca B.; Ribeiro, M.: Experience report: evaluating the effectiveness of decision trees for detecting code smells. In: 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE), pp. 261–269(2015)
72. Fontana, F.A.; Zanoni, M.; Marino, A.; Mantyla,M.V.: Code smell detection: towards a machine learning-based approach. In: 2013 IEEE International Conference on Software Maintenance pp. 396–399 (2013)
73. Maiga, A.; Ali, N.; Bhattacharya, N.; Sabane, A.; Gueheneuc, Y.G.; Aimeur, E.: SMURF: A SVM-based incremental anti-pattern detection approach. In: 2012 19th Working Conference on Reverse Engineering, pp. 466–475 (2012)
74. Khomh, F.; Vaucher, S.; Guéhéneuc, Y.-G.; Sahraoui, H.: BDTEX: a GQM-based Bayesian approach for the detection of antipatterns. J. Syst. Softw. **84**(4), 559–572 (2011)
75. Bryton, S.; e Abreu, F.B.; Monteiro, M.: Reducing subjectivity in code smells detection: Experimenting with the long method. In: 2010 Seventh International Conference on the Quality of Information and Communications Technology, pp. 337–342 (2010)
76. Kreimer, J.: Adaptive detection of design flaws. Electron. Notes Theor. Comput. Sci. **141**(4), 117–136 (2005)
77. Yang, J.; Hotta, K.; Higo, Y.; Igaki, H.; Kusumoto, S.: Classification model for code clones based on machine learning. Empir. Softw. Eng. **20**(4), 1095–1125 (2015)
78. Wang, X.; Dang, Y.; Zhang, L.; Zhang, D.; Lan, E.; Mei, H.: Can I clone this piece of code here?. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ACM, pp. 170–179 (2012)
79. Hassaine, S.; Khomh, F.; Guéhéneuc, Y.G.; Hamel, S.: IDS: An immune-inspired approach for the detection of software design smells. In: 2010 Seventh International Conference on the Quality of Information and Communications Technology, IEEE, pp. 343–348 (2010)
80. Vaucher, S.; Khomh, F.; Moha, N.; Guéhéneuc, Y.G.: Tracking design smells: Lessons from a study of god classes. In: 2009 16th Working Conference on Reverse Engineering, IEEE, pp. 145–154 (2009)
81. Fowler, M.; Beck, K.; Brant, J.; Opdyke, W.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston (1999)
82. Olbrich, S.; Cruzes, D.S.; Basili, V.; Zazworka, N.: The evolution and impact of code smells: a case study of two open source systems. In: 2009 3rd International Symposium on Empirical Software Engineering and Measurement, pp. 390–400 (2009)
83. Peters, R.; Zaidman,a.: Evaluating the lifespan of code smells using software repository mining. In: 2012 16th European Conference on Software Maintenance and Reengineering, pp. 411–416 (2012)
84. Tufano, M et al.: An empirical investigation into the nature of test smells. In: 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 4–15. (2016)
85. Tufano, M.; et al.: When and why your code starts to smell bad (and whether the smells go away). IEEE Trans. Softw. Eng. **43**(11), 1063–1088 (2017)
86. Taibi, D.; Janes, A.; Lenarduzzi, V.: How developers perceive smells in source code: a replicated study. Inf. Softw. Technol. **92**, 223–235 (2017)
87. Yamashita, A.; Moonen, L.: Do code smells reflect important maintainability aspects?. In: 2012 28th IEEE international conference on software maintenance (ICSM), pp. 306–315 (2012)
88. Palomba, F.; Zaidman, A.; Oliveto, R.; De Lucia, A.: An exploratory study on the relationship between changes and refactoring. In: 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), pp. 176–185 (2017)
89. Palomba, F.; Bavota, G.; Di Penta, M.; Fasano, F.; Oliveto, R.; De Lucia, A.: On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. Empir. Softw. Eng. **23**, 1–34 (2017)
90. Palomba, F.; Bavota, G.; Di Penta, M.; Oliveto, R.; De Lucia, A.: Do they really smell bad? a study on developers' perception of bad code smells. In: 2014 IEEE International Conference on Software Maintenance and Evolution, pp. 101–110 (2014)
91. Tamburri, D.A.; Palomba, F.; Serebrenik, A.; Zaidman, A.: Discovering community patterns in open-source: a systematic approach and its evaluation. Empir. Softw. Eng. **24**, 1–49 (2018)
92. Rapu, D.; Ducasse, S.; Girba, T.; Marinescu, R.: Using history information to improve design flaws detection. In: Eighth European Conference on Software Maintenance and Reengineering, CSMR Proceedings, pp. 223–232 (2004)
93. Palomba, F.; Panichella, A.; De Lucia, A.; Oliveto, R.; Zaidman, A.: A textual-based technique for smell detection. In: 2016 IEEE 24th International Conference on Program Comprehension (ICPC) pp. 1–10 (2016)
94. Sahin, D.; Kessentini, M.; Bechikh, S.; Deb, K.: Code-smell detection as a bilevel problem. ACM Trans. Softw. Eng. Methodol. **24**(1), 1–44 (2014)
95. Ozgur, A.: İstanbul: Boğaziçi University, Supervised and unsupervised machine learning techniques for text document categorization (2004)
96. Chaovalit, P.; Zhou, L.: Movie review mining: a comparison between supervised and unsupervised classification approaches. In: Proceedings of the 38th annual Hawaii international conference on system sciences, IEEE, pp. 112c (2005)
97. Kotsiantis, S.B.; Zaharakis, I.; Pintelas, P.: Supervised machine learning: A review of classification techniques. Emerg. Artif. Intell. Appl. Comput. Eng. **160**, 3–24 (2007)
98. Kotsiantis, S.B.; Zaharakis, I.D.; Pintelas, P.E.: Machine learning: a review of classification and combining techniques. Artif. Intell. Rev. **26**(3), 159–190 (2006)

99. Stamelos, I.; Angelis, L.; Oikonomou, A.; Bleris, G.L.: Code quality analysis in open source software development. Inf. Syst. J. **12**(1), 43–60 (2002)
100. Chidamber, S.R.; Kemerer, C.F.: Towards a metrics suite for object oriented design (1991)
101. Chidamber, S.R.; Kemerer, C.F.: A metrics suite for object oriented design. IEEE Trans. Softw. Eng. **20**(6), 476–493 (1994)
102. Desai, C.G.: Object oriented design metrics, frameworks and quality models. Inf. Sci. Technol. **3**(2), 66 (2014)
103. Shaheen, M.R.; Du Bousquet, L.: Survey of source code metrics for evaluating testability of object oriented systems (2010)
104. Abreu, B.E.; Carapua, R.: Object-oriented software engineering: Measuring and controlling the development process. In: 4th International Conference on Software Quality (ASQC). McLean, VA, USA (1994)
105. Radjenović, D.; Heričko, M.; Torkar, R.; Živkovič, A.: Software fault prediction metrics: a systematic literature review. Inf. Softw. Technol. **55**(8), 1397–1418 (2013)
106. Genero, M.; Piattini, M.; Calero, C.: A survey of metrics for UML class diagrams. J. Object Technol. **4**(9), 59–92 (2005)
107. Bansiya, J.; Davis, C.G.: A hierarchical model for object-oriented design quality assessment. IEEE Trans. Softw. Eng. **28**(1), 4–17 (2002)
108. Laradji, I.H.; Alshayeb, M.; Ghouti, L.: Software defect prediction using ensemble learning on selected features. Inf. Softw. Technol. **58**, 388–402 (2015)
109. Seliya, N.; Khoshgoftaar, T.M.; Van Hulse, J.: A study on the relationships of classifier performance metrics. In: 2009 21st IEEE International Conference On Tools With Artificial Intelligence, IEEE, pp. 59–66 (2004)
110. Cuadros-Rodríguez, L.; Pérez-Castaño, E.; Ruiz-Samblás, C.: Quality performance metrics in multivariate classification methods for qualitative analysis. TrAC Trends Anal. Chem. **80**, 612–624 (2016)
111. Wang, H.; Khoshgoftaar, T.M.; Seliya, N.: How many software metrics should be selected for defect prediction? In: Twenty-Fourth International FLAIRS Conference, Palm Beach, pp. 69–74 (2011)
112. Science, T.D.: 20 Popular Machine Learning Metrics. Part 1: Classification & Regression Evaluation Metrics (2019)
113. Hastie, T.; Tibshirani, R.; Friedman, J.: The elements of statistical learning. Springer series in statistics. Springer, Berlin (2001)
114. Candela, I.; Bavota, G.; Russo, B.; Oliveto, R.: Using cohesion and coupling for software remodularization: Is it enough? ACM Trans. Softw. Eng. Methodol. **25**(3), 24 (2016)
115. Simons, C. Singer, J.; White, D.R.: Search-based refactoring: Metrics are not enough. In: International Symposium on Search Based Software Engineering, pp. 47-61, Springer, Berlin (2015)
116. Chandrashekar, G.; Sahin, F.: A survey on feature selection methods. Comput. Electr. Eng. **40**(1), 16–28 (2014)
117. Tempero, E. et al.: The Qualitas Corpus: a curated collection of Java code for empirical studies. In: 2010 Asia Pacific Software Engineering Conference, pp. 336–345 (2010)
118. Maiga, A. et al.: Support vector machines for anti-pattern detection. In: 2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, pp. 278–281 (2012)
119. Guéhéneuc, Y.-G.; Antoniol, G.: Demima: a multilayered approach for design pattern identification. IEEE Trans. Softw. Eng. **34**(5), 667–684 (2008)
120. Guéhéneuc, Y.G.; Sahraoui, H.; Zaidi, F.: Fingerprinting design patterns. In: 11th Working Conference on Reverse Engineering, pp. 172–181 (2004)
121. Nguyen-Duc, A.; Cruzes, D.S.; Conradi, R.: The impact of global dispersion on coordination, team performance and software quality–A systematic literature review. Inf. Softw. Technol. **57**, 277–294 (2015)