International Journal of Software Engineering
and Knowledge Engineering
Vol. 29, No. 2 (2019) 285–316

DOI: 10.1142/S021819401950013X

Machine Learning Techniques for Code Smells Detection: A Systematic Mapping Study

Frederico Luiz Caram*, Bruno Rafael De Oliveira Rodrigues†,
Amadeu Silveira Campanelli‡ and Fernando Silva Parreiras§

LAIS Laboratory for Advanced Information Systems, FUMEC University
Av. Afonso Pena 3880, Belo Horizonte, MG, 30130009, Brazil
**fredcaram@fumec.edu.br*
†*brunorodrigues@fumec.edu.br*
‡*amadeu@fumec.edu.br*
§*fernando.parreiras@fumec.br*

Received 27 December 2017
Revised 22 January 2018
Accepted 8 August 2018

Code smells or bad smells are an accepted approach to identify design flaws in the source code. Although it has been explored by researchers, the interpretation of programmers is rather subjective. One way to deal with this subjectivity is to use machine learning techniques. This paper provides the reader with an overview of machine learning techniques and code smells found in the literature, aiming at determining which methods and practices are used when applying machine learning for code smells identification and which machine learning techniques have been used for code smells identification. A mapping study was used to identify the techniques used for each smell. We found that the Bloaters was the main kind of smell studied, addressed by 35% of the papers. The most commonly used technique was Genetic Algorithms (GA), used by 22.22% of the papers. Regarding the smells addressed by each technique, there was a high level of redundancy, in a way that the smells are covered by a wide range of algorithms. Nevertheless, Feature Envy stood out, being targeted by 63% of the techniques. When it comes to performance, the best average was provided by Decision Tree, followed by Random Forest, Semi-supervised and Support Vector Machine Classifier techniques. 5 out of the 25 analyzed smells were not handled by any machine learning techniques. Most of them focus on several code smells and in general there is no outperforming technique, except for a few specific smells. We also found a lack of comparable results due to the heterogeneity of the data sources and of the provided results. We recommend the pursuit of further empirical studies to assess the performance of these techniques in a standardized dataset to improve the comparison reliability and replicability.

Keywords: Machine learning; code smells; refactoring.

1. Introduction

One of the most costly operations involving software development is maintenance. It has been reported that above 75% of the total software cost is used for

§Corresponding author.

maintenance activities [6, 48]. An important factor about this is the quality of the written code, as it affects the readability, and hence the maintainability of the code [2], given that software maintainers spend around 60% of their time understanding the code they are working at [1]. Even in cautiously designed systems, the quality of the source code tends to degrade as the project evolves, given that the original design of a system is rarely prepared for every new requirement and quick changes are sometimes made by different people without properly adjusting the system structure [65]. Another factor is that developers also tend to focus on the addition of the new functions and bug fixes rather than improving software maintainability [68]. They also tend to overlook the code when it is apparently complex and when it seems not to be critical in maintaining the longevity of the software [56], one way to improve the software maintenance is avoiding code smells.

Code smells are code snippets with design problems, their presence in the code is difficult for the software maintenance and affects the quality of software. When a code smell is detected, it is suggested to do refactoring, to remove the code smells in the code that are refactoring to each other. The refactoring improves the code quality but it does not change the behave of system [13, 29]. Code smells provide heuristics for the identification of design flaws in the source code that makes software harder to evolve, comprehend and maintain. Each code smell examines a specific kind of system element (classes, methods, and so on) that can be evaluated by its characteristics [58]. As the software evolves, the number of code smells increases with time [12, 13].

The code smells provide guidelines to identify undesirable behavior and common coding mistakes, but they still depend on the interpretation of programmers [29], since there can be different interpretations according to each scenario and they may also be considered critical or not [67]. The definition of what is and what is not a code smell in a given context may not be in consensus among developers working in the same application [9, 25, 35], making their identification an error-prone and time-consuming task considering the size of commercial applications [56].

In order to reduce this subjective interpretation, automated approaches based on the source code were presented in the previous works [23, 26, 51, 62], but a relevant part of those approaches are based on code metrics [10, 14, 52, 55, 62]. These techniques use metrics and thresholds that are not consistent among them, leading to a growing number of false positives, not representing the real problem [25] since it does not consider information related to the context, domain, size and design of the system [21]. In this scenario, machine learning techniques can be used to capture this subjectivity. These techniques are utilized for a wide range of applications such as risk management [15], medicine [3], biology [38], financial markets [17], among others [20, 47]. And they can also be used for the identification of code smells in source code, providing more flexibility in comparison to the current metrics-based approaches [44]. The tools to identify code smells based on metrics do not analyze the past version of the code, so it is not allowed to understand the context of code smell,

while some code smells can be removed naturally in its evolution, others require more effort to resolve with refactoring [13].

Studies regarding the application of machine learning techniques are increasing, but each one uses different models and techniques to achieve this task [62]. Even with more studies appearing, it gets harder to find out the ones that perform better for each code smell. In other words, the real performance about the machine learning approaches applied to identify code smells. In order to address this gap, this study aims at understanding the methods and practices which are most frequently adopted in literature when applying machine learning for code smells identification and their reported performance. Thus, this paper answers the following research questions:

- **RQ1:** Which code smells are addressed by papers using machine learning techniques for code smells detection?
- **RQ2:** Which machine learning techniques are used to detect code smells?
- **RQ3:** Which machine learning techniques are the most used for each code smell?
- **RQ4:** Which machine learning techniques perform better for each code smell?

To answer these questions, we developed a mapping study on machine learning techniques and code smells found in the literature, covering papers from the introduction of anti-patterns in 1999 by Fowler and Beck[29] and including the papers published up to December 2016. The design flaws were categorized under the definition of the works defined by Fowler and Beck [29] and Brown *et al.* [8].

The study resulted in the classification of 26 papers out of 53 researched papers, separated and categorized by design flaws and applied techniques. We found that regarding F-measure: Association Rules techniques obtained better results on Speculative Generality, Divergent Change, Large Class and Long Method smells. Random Forest technique had better results on Large Class and Long Parameter List. On the other hand, Decision Tree is not good enough to identify Message Chains. While Naive Bayes Classifier presented the worst overall performance among the studied practices on Middle Man, Long Parameter List and Shotgun Surgery smells.

This paper is organized according to the following structure: Section 2 provides a background research about code smells, refactoring and machine learning techniques; Sec. 3 presents the related works of the area; Sec. 4 addresses the methodology used in this work; Sec. 5 displays the results of the study; Sec. 6 discusses the results; Sec. 7 presents the threats posed to the validity of the study and finally Sec. 8 shows the conclusion and provides suggestions for future work.

2. Background

2.1. *Code smells*

Code smells are the symptoms which can but not necessarily have to point to an actual issue. Therefore, they are not patterns to be avoided, but signals that require a

more thorough examination [69]. We follow a short description of each smell proposed by Fowler and Beck [29]:

- **Alternative Classes with Different Interfaces:** a case in which a class can operate with alternative classes but the interface of these classes is different.
- **Comments:** misuse of comments to compensate poor code structure.
- **Data Class:** a class that contains data but does not contain logic.
- **Data Clumps:** data items that usually appear together.
- **Divergent Change:** when a class needs to be changed every time another class is changed.
- **Duplicate Code:** code that does the same thing as another piece of code.
- **Feature Envy:** a method that is more interested in properties of other the classes than in the ones from its own class.
- **Inappropriate Intimacy:** when two classes are tightly coupled.
- **Incomplete Library Class:** when the software uses an incomplete library.
- **Large Class:** a class that tries to do a load of things, having plenty of instance variables or methods.
- **Lazy Class:** a class that is not doing enough and should be removed.
- **Long Method:** a method that is long, so it is hard to understand, change or extend.
- **Long Parameter List:** a parameter that is long and difficult to represent.
- **Message Chains:** a chain of calls from one object to another, without adding any new behavior.
- **Middle Man:** when a class delegates a great deal of its behavior to another class.
- **Parallel Inheritance Hierarchies:** a situation where two parallel class hierarchies exist and are related.
- **Primitive Obsession:** represents the usage of primitives instead of small classes, making it less meaningful and reusable.
- **Refused Bequest:** represents a child class that does not fully support its parent implementation.
- **Shotgun Surgery:** when a class change requires a broadcast changing of other classes.
- **Speculative Generality:** when unnecessary code is created anticipating future changes on software.
- **Switch Statements:** usage of type codes or run-time class type detection instead of polymorphism.
- **Temporary Field:** the class has a variable which is only used in specific situations.

Mantyla *et al.* [50] categorized code smells into eight categories as follows:

- **The Bloaters:** represent something that has grown so large that it cannot be effectively handled. This category covers the following smells: Long-Method; Large Class; Primitive Obsession; Long Parameter List; and Data Clumps.

- **The Object-Orientation Abusers:** represent code that does not exploit the possibilities of Object-Oriented Design. The following smells are included in this category: Switch Statements; Temporary Fields; Refused Bequest; Alternative Classes with Different Interfaces; Parallel Inheritance Hierarchies.
- **The Change Preventers:** smells that prevent or hinder the changing or further development of the system. This category is composed by: Shotgun Surgery and Divergent Change.
- **The Dispensables:** represent something that is unnecessary and should be removed from the code. Represented by the smells: Lazy class; Data class; Duplicated Code and Speculative Generality.
- **The Encapsulators:** smells that deal with data communication or encapsulation, including Message Chains and Middle Man.
- **The Couplers:** smells that increase the coupling of the system, being composed of Feature Envy and Inappropriate Intimacy.
- **Others:** smells that do not fit in any of the previous categories and are not comparable, such as: Incomplete Library Class and Comments.

Code smells can also be considered as symptoms of a design level flaw, also known as anti-patterns [55]. The anti-patterns concept was introduced by Brown *et al.* [8] defining it as a literary form that describes a recurrent solution to a problem that generates decidedly negative consequences. The studies also use approaches to identify the code anti-patterns instead of code smells, since they describe more generic flaws, in this study we use three of them:

- **The Blob:** corresponds to a large controller class that depends on data stored in surrounding data classes. A large class declares fields and methods with a low cohesion. A controller class monopolizes the processing done by a system, takes the main decisions, and directs the processing of other classes.
- **The Functional Decomposition:** consists of a main class in which inheritance and polymorphism are scarcely used, associated with small classes, which declare a great deal of private fields and implement sparse methods.
- **The Spaghetti Code:** classes with no structure, declaring long methods with no parameters, and utilizing global variables for processing. Names of classes and methods may suggest procedural programming.

2.2. *Machine learning*

Machine learning techniques can be categorized in three ways: supervised, unsupervised and semi-supervised. If instances are given with known labels (the human annotated correct output) then the learning is called supervised, otherwise, when the instances are unlabeled, it is unsupervised learning [37]. There is also a hybrid approach, which is the semi-supervised learning that uses both labeled and unlabeled data to perform an otherwise supervised learning or unsupervised learning task [74].

Supervised methods [44] are the leading approach used for code smells identification [22]. The following supervised approaches were identified by Kotsiantis [44] in his literature review:

- **Decision Trees:** Decision Trees are trees that classify instances by sorting them based on feature values. Each node in a Decision Tree represents a feature in an instance to be classified, and each branch represents a value that the node can assume. Instances are classified starting at the root node and sorted based on their feature values.
- **Learning Set of Rules:** Decision Trees can be translated into a set of rules by creating a separate rule for each path from the root to a leaf in the tree. However, rules can also be induced from training data using a variety of rule-based algorithms.
- **Single layered perceptrons:** Uses a single layer of weights to define a linearly separable binary classification.
- **Multi layered perceptrons (Artificial Neural Network):** Created to solve nonlinear classification problems that cannot be solved by a single layer. A multilayer neural network consists of large number of units (neurons) joined together in a pattern of connections.
- **Radial Basis Function (RBF) network:** An RBF network is a three-layer feedback network, in which each hidden unit implements a radial activation function and each output unit implements a weighted sum of hidden units outputs.
- **Naive Bayes:** Naive Bayesian networks (NB) are simple Bayesian networks which are composed of graphs with only one unobserved node and a chain of children observed nodes, with an assumption of state independence between child nodes and their parent. The Naive Bayes is based on estimating the probabilities of the unobserved node, based on the observed ones.
- **Bayesian networks:** A Bayesian Network (BN) is a graph-based model that establishes a probability relationship among a set of known variables. The Bayesian network structure is a graph containing nodes linked with its features. The features must be conditionally independent from their non-descendants in relation to its parents.
- **Instance Based learning:** Instance-based learning algorithms are statistically-based and lazy-learning algorithms, as they delay the induction or generalization process until classification is performed.
- **Support Vector Machines (SVM):** SVMs are based on the notion of a "margin" in either side of a hyperplane separating two features. Its optimizing objective is to increase the margin and create the largest distance between features in the hyperplane. The complexity is unaffected by the number of features. So SVMs are suited to deal with learning tasks where the number of features is large with respect to the number of training instances.

Unsupervised learning is composed mainly of clustering techniques. The clustering objective is to develop an automatic algorithm that discovers the natural groupings in the unlabeled data [37]. Clustering algorithms can be broadly divided into two groups: hierarchical and partitional. Hierarchical clustering algorithms recursively find nested clusters, while the partitional clustering finds the clusters simultaneously. In semi-supervised clustering, instead of specifying the class labels, constraints are specified, as a weaker way of encoding the labeled data. Semi-supervised learning can be applied in place supervised learning, using unlabeled data for training [36]. Semi-supervised learning has been applied to natural language processing (word sense disambiguation, document categorization, named entity classification, sentiment analysis, machine translation), computer vision (object recognition, image segmentation), bio-informatics (protein function prediction), and cognitive psychology [74], and also to address code smell identification problems [22].

Other algorithms like genetic algorithms (GA) can be used to identify code smells. The GAs are algorithms that try to imitate natural selection based on biological mechanisms [57]. They work with concept of genome and three operators: Reproduction, crossover and mutation, proving robust search in complex spaces, which they find the optimal solution [33, 45, 57].

3. Related Work

Rasool and Arshad [62] elaborated a literature review to identify state-of-the-art tools, which are used for mining the code smells based on the source code of different software projects. Its identified that earlier code smell detection techniques were focused on static source code analysis methods for detecting code smells, afterwards it shifted to a combination of static and dynamic source code analysis methods. Recently, search-based code smell detection techniques obtained attention by applying code metrics and machine learning methods for the identification. A large number of code smell detection techniques used source code metrics for the detection, computing metrics from source code or from third-party tools, usually aiming at Feature Envy, Data Class, Large Class, Long Method, and Long Parameter List code smells, while alternative classes with different interfaces and Incomplete Library Class were not covered. In regard to the tools, many apply different object-oriented source code metrics to detect a large number of code smells. They presented high language dependency for the detection of code smells, given that 38 out of the 46 tools are focused on Java language. The same pattern repeated regarding the experiments. The performance of the tools varied depending on the studied smell, the same happened in the use of different tools for the same smell. They claim those differences due to the selection of different metrics and thresholds by the different techniques and also criticize the unavailability of standard benchmark systems for comparing results of code smell detection tools. The dependence on code metrics also made it harder to reuse the techniques to detect different code smells, the accuracy

commonly relied on the selection of threshold values and may be deceptive. 30% of the tools spotlight the accuracy, that is, precision and recall, of their technique or tool. The study provided an insight on the evolution of the techniques used for code smell and the current state of the tools and techniques, identifying research gaps and opportunities, exposing that the machine learning techniques have been generating growing interest through the researches. Our study aims to cover the machine learning techniques, to understand how they are being used and which techniques fit and perform better for each code smell. Apart from the given study, that focuses on general techniques and tools that were used for code smells identification, we did a mapping study focusing on the machine learning techniques that were applied to code smells and their performance.

Fernandes *et al.* [22] performed a systematic review to identify and document all tools reported and used in the literature for bad smell detection. From the 22 bad smells defined by Fowler and Beck [29], they identified tools to detect 20 of them, the exceptions are Alternative Classes with Different Interfaces and Incomplete Library Class. While Duplicated Code and Large Class were the most targeted smells, more than 40% of the tools target at least one of these bad smells. In addition, there were also tools to identify 41 smells defined by other authors. From the analyzed tools, 30 of them are plug-ins, 30 are standalone applications and four of them are available as both. They found a concentration of proposed tools for three languages: Java, C, and C++. But only one out of the top 10 used languages was not covered by any tool. The authors developed a comparison among the four selected tools: inFusion; JDeodorant; PMD and JSpirit, and two code smells: Large Class and Long Method, selected because they are the most covered smells, for Duplicated Code, it was hard to quantify its results. Each tool was tested against the projects: JUnit, to check agreement between the tools, and MobileMedia for agreement, recall and precision. Regarding recall, PMD and JSpIRIT provided the highest results achieving 50% and 67%, respectively. When analyzing precision, inFusion and PMD had the highest values achieving 100%. This study contributed for the automatic code smell detection, by cataloguing the tools used for code smells detection and also the situations in which they can be used. However, this study differs from ours, since it focuses on tools for code smells detection and not on the techniques, it provides little information about the used techniques, as some tools do not provide that information, whereas our study focuses on the machine learning techniques, excluding studies that rely only in predefined or user-defined metrics.

Rattan *et al.* [63] developed a systematic literature review aiming at identifying and classifying the existing literature about clone detection, clone management, semantic clone detection and model-based clone detection techniques. In order to do so, 213 studies were reviewed, where 100 were found to be research studies of software clone detection. The studies generally used an intermediate code representation, with different granularity, Abstract Syntax Trees (ASTs) or Parse trees, source code or text and Regularized tokens are the most frequently used. Regarding the matching technique: metrics/feature vectors clustering, Suffix Tree-based token by

token, substring/subtree model comparison and Dynamic programming were the most common approaches. The authors concluded that clones definition are still unclear and that there is a lack of empirical studies regarding the harmfulness of clones. They also concluded that it can be used as principled re-engineering technique and be beneficial as it is not easy to refactor all the clones due to cost/risk associated with refactoring. It is suggested that instead of removing clones, we should have proper clone management facilities. This study is similar to ours as it also focuses on a code-smell identification and the techniques used for that, but it differs from our study since it focuses only on one code smell and uses a broader scope for the techniques and it also contemplates any automated identification task and not only machine learning techniques.

Al Dallal [4] reports a systematic literature review that identifies the state-of-the-art techniques regarding the identification of refactoring opportunities, assessing and discussing the collected and reported findings. The studies considered 22 refactoring opportunities; 20 of them are among the 72 activities identified by [23, 29], and two were proposed by others. It identified that Quality metrics-oriented, Precondition-oriented and Clustering-oriented were the preponderant techniques. For the empirical evaluation, intuition-based techniques were frequently used, followed by Quality-based and Mutation-based evaluations. The studies in general used Java open source projects for their empirical experiments. The author found that the studies focuse more on Move Method, Extract Class, and Extract Method than in the other refactoring activities, justifying their importance for the industry, while 72.2% of the refactoring activities proposed by Fowler were not considered by any study. It was also found that most of the studies lack empirical data to support their techniques. This study is related to ours, since the techniques used for refactoring and the techniques for the code smells identification share common features, such as metrics and techniques aiming at their identification, but it is different from ours in its nature, as we focus on identifying the code smells while it focuses on identifying refactorings opportunities.

4. Research Method

This study performed a mapping study on the usage of machine learning techniques for code smells identification. Mapping studies are based on a clear search strategy, that ensures the rigor, completeness and a reproducible process, focusing on the identification, evaluation and interpretation of the available research that is relevant to a particular question [41].

The process adopted for mapping study was based on the work of Kitchenham *et al.* [42] that includes three phases: planning, conduction and documentation. The following subsections describe the steps taken in the planning and conduction phases. The documentation phase is addressed in Secs. 5 and 6.

4.1. *Planning*

We developed a protocol according to the guidelines provided by Kitchenham *et al.* [42] in order to ensure that the research was executed in a planned way and not driven by the expectations of the researcher. In the protocol, the following items were documented as part of the planning for the study: research objectives; research questions; search strategy; study selection, quality assessment of the studies; data extraction; and data synthesis and aggregation.

4.2. *Research questions*

This research focuses on the identification machine learning techniques used for code smells detection and in order to accomplish that the following questions must be answered:

- **RQ1: Which code smells are addressed by papers using machine learning techniques for code smells detection?** RQ1 focuses on the most common code smells detected by studies utilizing machine learning techniques. The answer to this research question can help in identifying code smells that have not been explored in existing research and those that can be opportunities for the future work.
- **RQ2: Which machine learning techniques are used to detect code smells?** RQ2 aims in the identification of machine learning techniques commonly used to detect code smells in the literature. This information can be useful to determine the most popular machine learning techniques used in the code smells domain and also to generate a list of potential machine learning techniques that were not explored for code smells detection.
- **RQ3: Which machine learning techniques are the most used for each code smell?** RQ3 is concerned with the relationship between the machine learning techniques and the code smells detected by these techniques. The outcome of this question can provide insights on the association between machine learning techniques and types of code smells. It could also lead to the identification of gaps in research pointing to potential machine learning techniques that can be used for similar code smells but have not been used for this intent.
- **RQ4: Which machine learning techniques perform better for each code smell?** RQ4 compares the performance of different machine learning techniques for code smell detection. The answer to this question can help the researchers to define which machine learning techniques to use when studying different code smells.

4.3. *Search strategy*

In order to find relevant papers to achieve the goals of this study, we conducted searches based on the recommendations by Kitchenham *et al.* [42]. In the first step, we used our personal experience to define a list of the most relevant journals and conferences as summarized in Table 1, which was used to establish the quasi-gold

standard (QGS). The QGS for this mapping study is composed of 21 papers. The following digital libraries were used in the automated searches: ACM, IEEE, Science Direct and Wiley.

Table 1. List of conferences and journals used in the manual search.

Conference/Journal Title
- Empirical Software Engineering
- Expert Systems with Applications
- International Conference on Software Engineering (ICSE)
- IEEE Transactions on Software Engineering
- International Conference on Frontiers in Intelligent Computing: Theory and Applications
- International Conference on Software Maintenance (ICSM)
- International Symposium on Software Reliability Engineering (ISSRE)
- Journal of Software Maintenance and Evolution
- Journal of Software: Evolution and Process
- Journal of Systems and Software
- Knowledge-Based Systems

The definition of the search string followed these steps:

(1) Derived major terms for machine learning and code smells from research papers.
(2) Broke major terms down into smaller terms the machine learning related terms were based on [71].
(3) Identified alternative spellings or synonyms for the smaller terms.
(4) Checked the search terms in known relevant papers.
(5) Used Boolean operator OR combined alternative spellings and synonyms. Used Boolean operator AND linked machine learning and code smells terms.

This is the search string used for papers retrieval in automated searches: "("code smell" OR "bad smell") [29] AND ("learning" OR "data mining" OR "artificial intelligence" OR "pattern recognition" OR "case based reasoning" OR "decision tree" OR "regression tree" OR "classification tree" OR "neural net" OR "genetic programming" OR "genetic algorithm" OR "Bayesian belief network" OR "Bayesian net" OR "association rule" OR "support vector machine" OR "support vector regression")" [71]. The automatic search returned 1021 papers distributed among the digital libraries as summarized in Fig. 1.

The search performance was measured using the "quasi-sensitivity" which is the recall of the search. The result had a sensitivity of 85% higher than the threshold of 70% to 80% proposed by Zhang *et al.* [72]. Based on this result, we proceeded to the next phase of the mapping study.

4.4. *Studies selection*

The selection criteria target at filtering relevant studies to the research questions [42]. We defined three inclusion and three exclusion criteria.

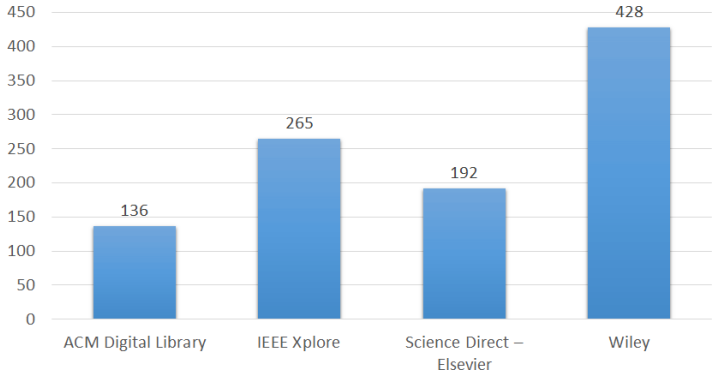

Fig. 1. Distribution of papers found in automated search by digital library.

Inclusion criteria:

- Papers from computer science, information system or software development fields.
- Papers describing the use of machine learning techniques applied to code smells detection.
- The studies should be peer-reviewed.
- The studies should be published in English.

Exclusion criteria:

- Remove duplicates.
- Remove studies in which the code smells detection is done manually.
- Remove works that were not completed.
- Remove non-empirical studies.

The selection process consisted of five stages as shown below and it was administered by three of the authors. Each step was peer-reviewed by graduated students, and the results were compared and discussed in order to reach a consensus.

(1) The databases search returns 1021 papers.
(2) After the exclusion of duplicate paper 683 papers left.
(3) Evaluated the type of review, field of the research and publication language 286 papers left.
(4) Removed studies unrelated to machine learning techniques for code smells identification and also non-empirical studies 155 papers left.
(5) Removed papers based on title and abstract 53 papers left.

In the first stage of the selection process, we removed 338 duplicated papers and moved 683 papers to stage 2. Stage 2 evaluated the type of review, field of the research and publication language. This stage selected 286 papers to move to the

next part of the process. The studies were then filtered by title in stage 3, removing studies unrelated to the usage of machine learning techniques for code smells identification and also non-empirical studies resulting in 155 papers. In stage 4, we applied the same criteria as in stage 3 but the filtering was based on the abstract of the studies. Stage 4 selected 53 papers to continue in the review process going through quality assessment and classification.

4.5. *Quality assessment*

In this mapping study, we used the quality assessment to have a better insight about the selected papers to exclude papers for the final data extraction and analysis. Our intention was to use the detailed quality information about the paper during analysis and also to identify which percentage of the papers would have comparable results.

The quality assessment questions used in our study were based on the quality checklist defined by Dybå and Dingsøyr [18]:

- **QA1:** Is the paper based on research (or is it merely a lessons learned report based on expert opinion)?
- **QA2:** Are the aims of the research defined explicitly?
- **QA3:** Is there experimental design appropriate and justifiable?
- **QA4:** Is the proposed estimation method comparable with other methods?
- **QA5:** Are the findings of study stated and supported by the reported results?
- **QA6:** Does the study add value to the academy or to the practice communities?

The quality of the papers was assessed by three of the researchers during data extraction. All of them performed the quality assessment checklist for each of the papers. In order to reach a consensus a Delphi method [16] was used. Initially, each participant received a spreadsheet containing the selected papers and the quality assessment questions, which they had to fill with either yes or no. In each round a paper was selected and the researchers answers to each of the quality questions was compared, in case of disagreement the participants had to justify and discuss their answers, then another round would be executed. This process was repeated until a consensus was achieved.

4.6. *Data extraction and classification*

The data extraction process was conducted by three of the authors. Each author read the papers and provided values for the items available in the classification form. The data was reconciled using discussion and moderation to achieve an agreement between the reviewers.

The classifications used in this phase were Fowler and Beck [29], Rasool and Arshad [62], Fernandes *et al.* [22]:

- Data source: The projects used for the technique assessment;

- Language: The language of the used project;
- Metrics: Which metrics were taken into consideration;
- Baseline: The baseline against which the technique performance was compared;
- Addressed Code Smells: The code smells addressed by the paper.

5. Results

In this section, we present the results of the systematic mapping. Firstly, we show an overview of the selected papers and then answers to the research questions and their findings.

5.1. *Overview*

In this study, we identified 26 papers that used machine learning techniques for code smell identification. They were published between 1999 to 2016 and they used experimentation as methodology. 65% of the papers were published in conferences and the other 37% were published in journals. Five publications were represented by more than one publication venue: Annual Conference on Genetic and Evolutionary Computation; International Conference on Software Engineering; Journal of Systems and Software; Expert Systems with Applications; Journal of Software: Evolution and Process. These publications represented almost half of the selected papers as shown in Table 2.

Table 2. Papers by publication.

Publication	No. of studies
Annual Conference on Genetic and Evolutionary Computation	3
International Conference on Software Engineering	3
Journal of Systems and Software	3
Expert Systems with Applications	2
Journal of Software: Evolution and Process	2
Others	13

In Fig. 2, it is possible to see a trend in the publications over the years. Figure 2 shows a greater interest in studies identifying code smell using machine learning techniques in 2015 and 2016 than between 2002 and 2014. The number of researches in the last two years overcame the numbers of the previous years. It is worth

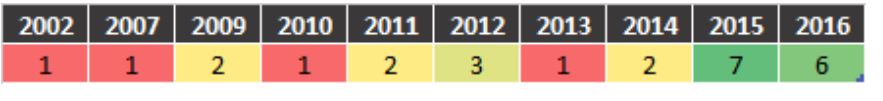

Fig. 2. Number of papers by year.

remembering that the code smell term was used by Kent Beck in the late 1990s and the research about code smell was focused on other techniques as metrics. Detection of code smells based on machine learning techniques has not been extensively explored [27].

All papers selected in this study identified code smells analyzing projects developed with Java language. Out of 26 papers, 23 used open source projects, while three used private data source. We identified 88 open source projects as dataset used in research, the dataset more frequently used was the Xerces, followed by JHotDraw, Eclipse Core and ArgoUML. Among the 88 datasets, 62 of them were used once (see Table 3).

Table 3. Open source projects adopted.

Software	No. of papers	Software	No. of papers
Xerces	6	Apache Commons Logging	1
JHotDraw	5	JDI-Ford	1
Eclipse Core	5	Apache Derby	1
ArgoUML	4	JFreeChart	1
JFreeChart	3	Apache James Mime4j	1
GanttProject	3	JRDF	1
Apache Cassandra	2	Apache Tomcat	1
Rhino	2	Maven	1
Qualitas Corpus	2	ApacheAnt	1
GanttProject	2	Pixelitor	1
jEdit	1	ApacheAnt	1
Ant	1	sapphire	1
Log4J	1	Android API (framework-opt-telephony)	1
And Engine	1	XWorks	1
JabRef	1	BCEL	1
Apache Commons Codec	1	Jboss	1
Android API (tool-base)	1	Closure Compiler	1
Apache Commons IO	1	JDK	1
nebula.widgets.nattable	1	dltk.core	1
Apache Commons Lang	1	Android API (sdk)	1
Xerces	1	Android API (frameworks-base)	1
JGraphx	1	Aardvark	1
egit	1	platform.resources	1
JHotDraw	1	Gitblit	1
FindBugs	1	Ant-Apache	1
jUnit	1	Google Guava	1
FreeMind	1	ANTLR	1
Lucene	1	graphiti	1
GanttAzureus	1	Xom	1
Mongo DB	1	Guava	1
Android API (frameworks-support)	1	Apache Ant	1
Nutch	1	Hibernate	1

5.2. *Which code smells are addressed by papers using machine learning techniques for code smells detection?*

In this section, in addition to Fowler and Beck [29] smells and Brown *et al.* [8] anti-patterns, we also included a classification called "others", meant to capture smells defined by other authors and code-flaws not related to a specific smell, since the authors aim at code metrics optimization instead of a specific smell.

Comparing the studied code-related design flaws using machine learning, the ones with higher occurrence were Feature Envy smell and BLOBs, both studied by five papers, followed by Long Methods that showed up in four papers. While Comments, Primitive Obsession, Refused Bequest, Alternative Classes with Different Interfaces and Incomplete Library Class were not addressed by any of the studied papers. The distribution of the smells is displayed in Fig. 3. Except for the Duplicated Code, which is one of the most studied smells, the other smells are coherent with the ones identified in [73]. The category Others, composed mainly of optimization in the code metrics, appeared eight times, using alternatives to the traditional code smells.

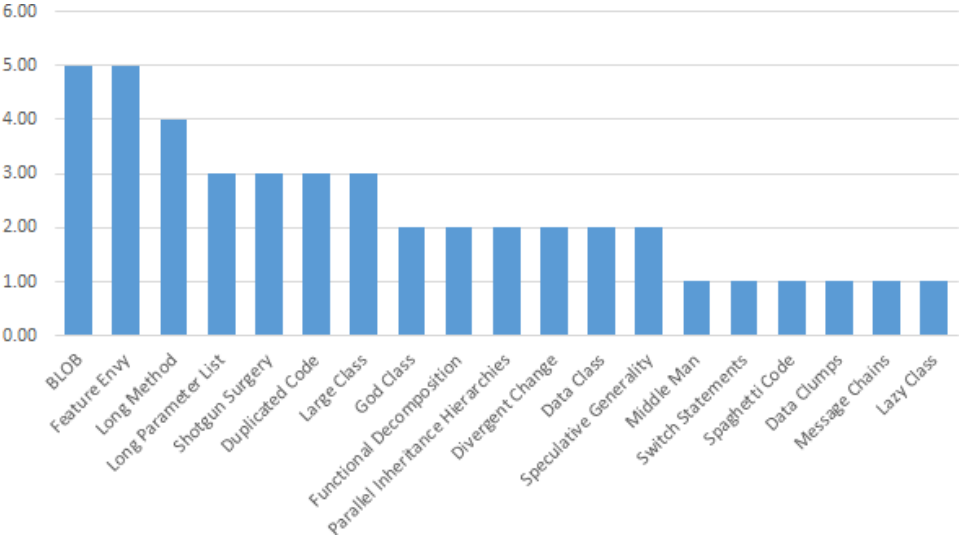

Fig. 3. Number of papers by code smell.

When grouping by the classification defined by Mantyla *et al.* [50] it is possible to observe that the main focus regarding the addressed type of smell is the Bloaters representing 35% of the studied smells. As detailed in Fig. 4.

When analyzing the smells studied together, one of the strongest co-occurrences is between BLOB, Spaghetti Code and Functional Decomposition. One of the main reasons is that they use the anti-patterns definition by Brown *et al.* [8], while the

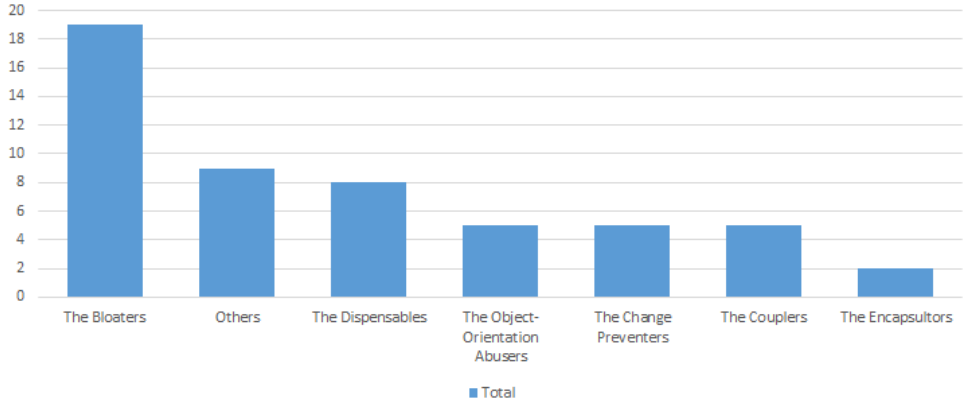

Fig. 4. Number of papers by code smell type.

others are defined by Fowler and Beck [29]. Long Parameter List, Large Class and Long class also presented a high correlation, a possible explanation is that they share the same type, what makes it easier to apply the same kind of algorithm to all of them. The co-occurrence graph is shown in detail in Fig. 5.

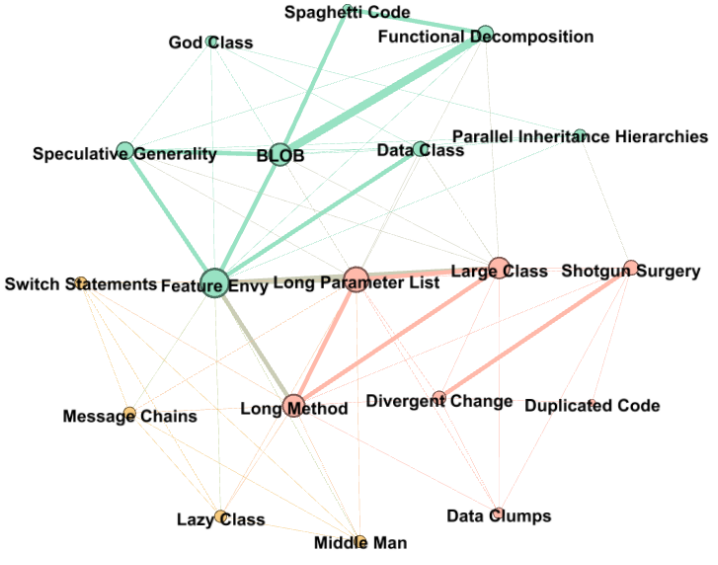

Fig. 5. Graph representing which smells were assessed in the same paper.

5.3. *Which machine learning techniques are used to detect code smells?*

The leading technique in the analyzed papers was the GA which appeared eight times. This technique is used in search-based techniques and focuses on optimizing

one or more metrics by mutating and enhancing the code. It was followed by Naive Bayes Classifiers that appeared four times. Whereas approaches such as Linear discriminant analysis; Decision Tree; Support; Vector Machine; Directed Acyclic Graph; and Text-Based; showed up once. The distribution can be viewed in Fig. 6.

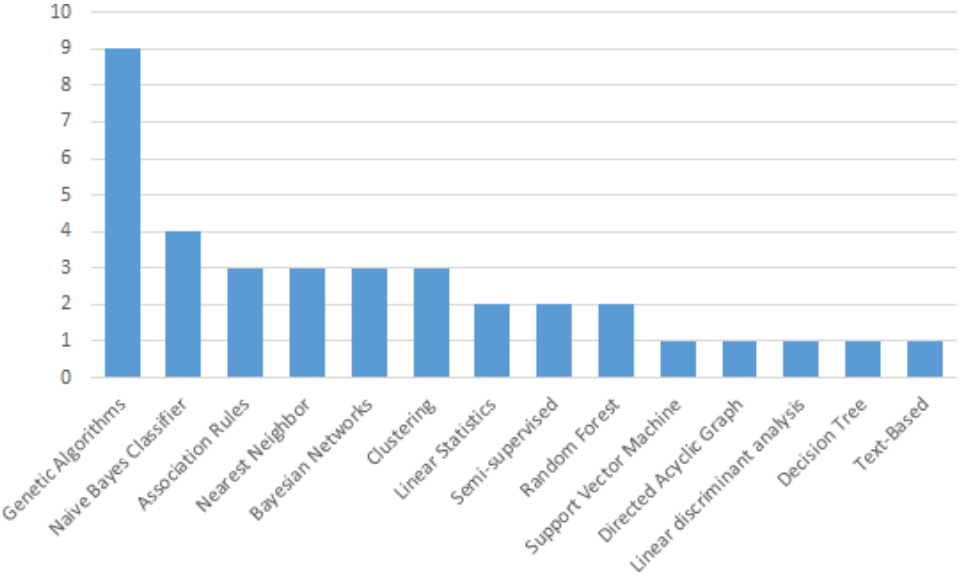

Fig. 6. Number of papers by machine learning technique.

Regarding the kind of technique used, the supervised techniques were the majority, being used in 32 tests (88%); semi-supervised and unsupervised techniques appeared 6% each. The results were expected since supervised tests are the most often used in researches [44].

5.4. *Which machine learning techniques are the most used for each code smell?*

The smells addressed by the techniques were coherent with the smells that appeared in the related papers [22], showing a high level of redundancy. The smell focused by most of the techniques was Feature Envy, which was aimed by nine techniques (64%). Followed by Long Method with eight techniques (57%) and Long Parameter List with seven (50%). From the smells targeted by the studied papers, the ones covered by less techniques were Speculative Generality, Spaghetti Code, Data Class, God Class, Parallel Inheritance and Divergent Changes with two techniques each (14%).

From a machine learning technique perspective, the Association Rules technique was the technique with the highest usage, covering 13 smells (59% of them), followed

by Linear Statistics with 11 smells (50%), Naive Bayes and Random Forest with 9 smells (43%). While Text-Based and Linear Discriminant Analysis with 1 smell (5%) are in the lower half.

When comparing the relationship between the code smell and the techniques, there was a relationship between the following techniques and the respective code smells: Association Rules for Divergent Change, God Class, Data Clumps, Parallel Inheritance, Shotgun Surgery and Speculative Generality; Bayesian Networks for Speculative Generality; Clustering for God Class and Parallel Inheritance Hierarchies; GAs for Spaghetti Code; Linear Statistics for Data Clumps and Semi Supervised for Speculative Generality. The relation between smells and techniques can be visualized in Fig. 7. When analyzing the smell categories, we found out the Association Rules technique with a relevant focus on the Change Preventers type of smell, as the usage of the technique focus on the relationship between methods and classes. The other smell types did not show any relevant relationship.

	Code Smells																			
Machine Learning Techiniques	**BLOB**	**Data Class**	**Data Clumps**	**Divergent Change**	**Duplicated Code**	**Feature Envy**	**Functional Decomposition**	**God Class**	**Large Class**	**Lazy Class**	**Long Method**	**Long Parameter List**	**Message Chains**	**Middle Man**	**Parallel Inheritance Hierarchies**	**Shotgun Surgery**	**Spaghetti Code**	**Speculative Generality**	**Switch Statements**
Association Rules	17%	25%	50%	67%	25%	8%	0%	50%	17%	0%	8%	10%	0%	0%	50%	50%	0%	50%	0%
Bayesian Networks	33%	0%	0%	0%	0%	0%	33%	0%	0%	0%	0%	0%	0%	0%	0%	0%	50%	0%	0%
Clustering	0%	0%	0%	0%	0%	0%	0%	50%	0%	0%	0%	0%	0%	0%	50%	25%	0%	0%	0%
Decision Tree	0%	0%	0%	0%	0%	8%	0%	0%	0%	14%	8%	10%	14%	14%	0%	0%	0%	0%	14%
Genetic Algorithms	33%	0%	0%	0%	25%	0%	33%	0%	0%	0%	0%	0%	0%	0%	0%	0%	50%	0%	0%
Linear discriminant analysis	0%	0%	0%	0%	0%	8%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Linear Statistics	0%	0%	50%	33%	0%	8%	0%	0%	17%	14%	15%	20%	14%	14%	0%	25%	0%	0%	14%
Naive Bayes Classifier	0%	25%	0%	0%	0%	23%	0%	0%	17%	29%	23%	20%	29%	29%	0%	0%	0%	0%	29%
Nearest Neighbor	0%	0%	0%	0%	25%	15%	0%	0%	0%	29%	15%	20%	29%	29%	0%	0%	0%	0%	29%
Random Forest	0%	25%	0%	0%	0%	15%	0%	0%	17%	14%	15%	10%	14%	14%	0%	0%	0%	0%	14%
Semi-supervised	17%	0%	0%	0%	25%	8%	33%	0%	17%	0%	0%	10%	0%	0%	0%	0%	0%	50%	0%
Support Vector Machine	0%	25%	0%	0%	0%	8%	0%	0%	17%	0%	8%	0%	0%	0%	0%	0%	0%	0%	0%
Text-Based	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	8%	0%	0%	0%	0%	0%	0%	0%	0%

Fig. 7. Relationship between techniques and code smells.

5.5. *Which machine learning techniques performs better for each code smell?*

One hardship we found when comparing the performance of the techniques is the lack of standardized data. 14 out of the 26 papers provided performance information. From those, 13 provided precision data, 12 provided recall values and 6 provided us with F-measures. We used the recall and precision data provided by the other to calculate their F-measures as well as to have a comparison measure. We also selected

measures that the code smells. The compared values are detailed in Tables A.1 and A.2.

In terms of f-measure the best average performance was provided by Decision Tree, followed by Random Forest, Semi-supervised and Nearest Neighbor techniques. While Text-Based, Linear Discriminant Analysis and Naive Bayes presented the worst performance overall between the studied practices, as demonstrated in Fig. 8.

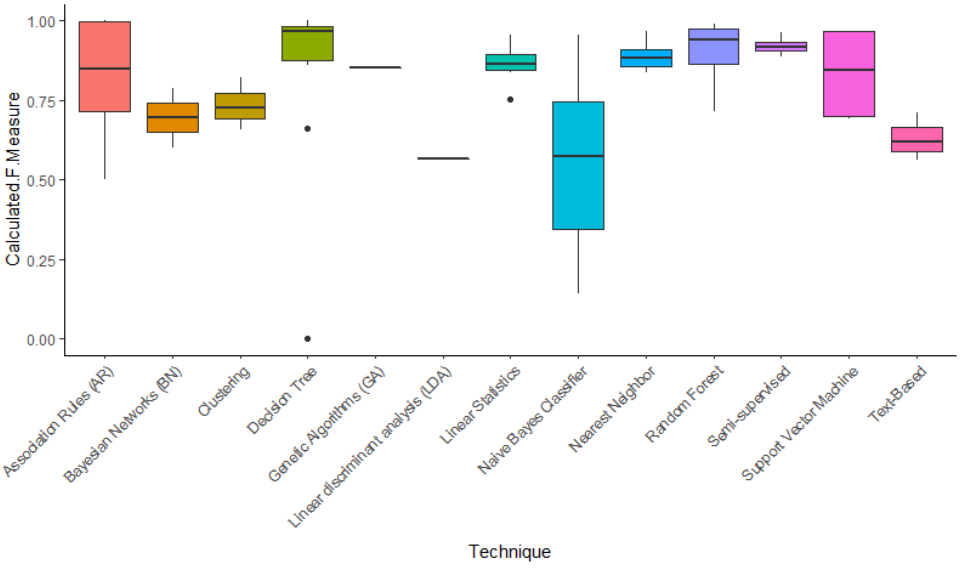

Fig. 8. Machine learning techniques f-measure box-plot.

When comparing the results by f-measure as demonstrated in Fig. 9, it is possible to notice that the Association Rules technique performed above the others for Divergent Changes and Speculative Generality smells, the ones covered by this technique. But it performs poorly for Duplicated Code and Feature Envy, when compared to the other techniques. Decision Tree was also the best performing technique for Middle Man and Shotgun Surgery smells, while Random Forest had an outstanding performance for Long parameter list and Semi-supervised techniques for Duplicated Code. Naive Bayes on the other side, performed poorly for Long Parameter List, Middle Man and Shotgun Surgery. In regard to the other smells, there was no outstanding technique.

When analyzing the techniques by precision, the Linear Discriminant Analysis technique presented the best average performance, followed respectively to its performance by Association Rules, Semi-supervised and Decision Tree. In this aspect the worst performing techniques were the Bayes-based techniques: Naive Bayes Classifier and Bayesian Networks. The results can be visualized in Fig. 10.

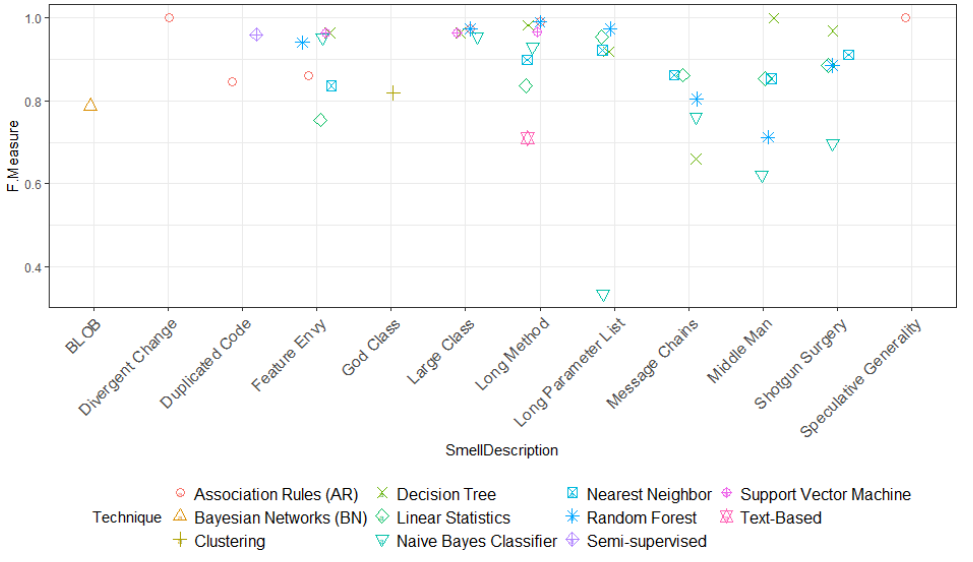

Fig. 9. F-Measure technique by code smell.

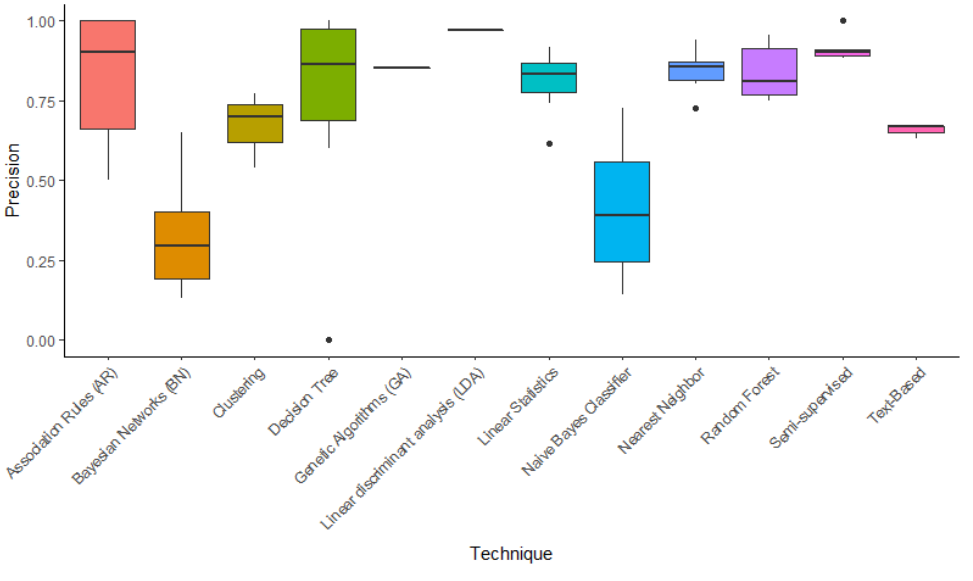

Fig. 10. Machine learning techniques precision box-plot.

Comparing the techniques by precision, Association Rules, also demonstrated an outstanding performance on Divergent Changes and Speculative Generality, where it is the only technique used. Semi-supervised techniques had an outstanding performance for Duplicated Code, Random Forest for FE

and Decision Tree for Lazy Class and Middle Man also worth mentioning. We had the Bayesian Networks and Naive Bayes Classifiers performing poorly than the other techniques on the smells. Those observations are displayed in Fig. 11.

Fig. 11. Precision technique by code smell.

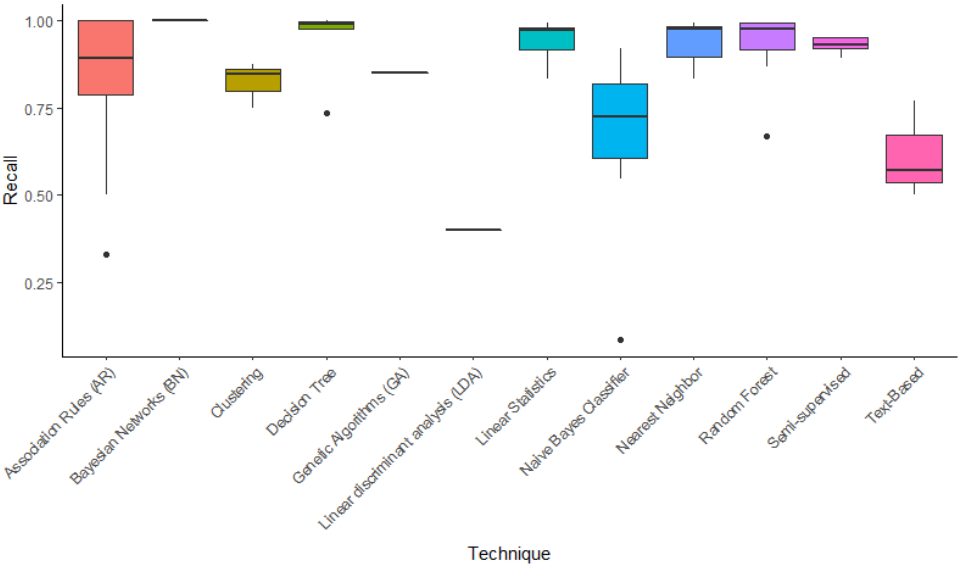

Fig. 12. Machine learning techniques recall box-plot.

When compared by recall, the techniques, in general, performed above 80%. The best performing techniques under these perspectives were: Bayesian Networks, Decision Tree, Random Forest, Nearest Neighbor, Linear statistics, Semi-supervised, GA and Clustering. While the worst performance came from Naive Bayes classifier, Text-Based and Linear Discriminant Analysis, as displayed in Fig. 12.

We assessed the technique by smells under a recall perspective as demonstrated in Fig. 13. As occurred in the previous perspective, we have the Naive Bayes classifier performing worst in general. Other techniques that displayed a bad performance were Random Forest for Middle Man, Decision Tree for Message Chain and Text-Based technique for Long Method.

Fig. 13. Recall technique by code smell.

6. Discussion

This research tried to comprehend the patterns regarding machine learning applied for code smells identification. The study covered the papers in the period of 1999 to 2016, although no paper was published on the matter for about 2 years after the publication of the code smells by Fowler and Beck [29] and it has been an active research topic for the last 2 years, as shown in Table 2.

By studying the smells it was possible to assert that contrary to Fowler and Beck [29], Rasool and Arshad [62], Fernandes *et al.* [22], Rattan *et al.* [63] that showed the Duplicated Code as the leading smell, the ones using machine learning showed more concern about Feature Envy, BLOB and Long Methods, the latter was also covered by code smell detecting tools as highlighted by Rasool and Arshad [62]. There is

increasingly focus on search based techniques, based on GA, coinciding with the results found by Rasool and Arshad [62].

Regarding the techniques, although techniques such as Naive Bayes and Nearest Neighbors have been used more often than the others due to their simplicity, on the overall we did not find any killer technique receiving more attention, corroborating with the study by Fernandes *et al.* [22], which found a high variance in the result when comparing the same technique for different code smells. However, when comparing the techniques for each code smell, we found that the following techniques are more used for the following specific smells: Association Rules for Divergent Change, God Class, Data Clumps, Parallel Inheritance, Shotgun Surgery and Speculative Generality; Bayesian Networks for Speculative Generality; Clustering for God Class and Parallel Inheritance Hierarchies; GAs are more used for Spaghetti Code; Linear Statistics for Data Clumps and Semi-supervised for Speculative Generality. We also found that although the machine learning techniques were used for 18 out of the 22 [29] smells, it still covers less smells than the existing smell detecting tools, that as stated by Rasool and Arshad [62] and Fernandes *et al.* [22] on their reviews covered 20 out of the 22 smells.

Comparing the performances we found that on average Decision Tree, followed by Random Forest, had the best performance, agreeing with the experiment by Fontana *et al.* [25] which found Decision Tree and Random Forest related algorithms to outperform others on smell identification tasks. Semi-supervised and Nearest Neighbor classifiers also slightly outperformed the remaining techniques, while Text-Based, Linear Discriminant Analysis and Naive Bayes presented the worst performance overall between the studied practices, going against the findings of Fontana *et al.* [25], Soltanifar *et al.* [66] that found the Bayes approaches performing well for class-related smells. There were also techniques that performed better for specific smells such as Association Rules for Divergent Changes and Speculative Generality, Decision Tree for Lazy class, Middle Man and Shotgun Surgery smells, Random Forest for Long parameter list and Semi-Supervised techniques for Duplicated Code. When comparing the same techniques, used for the same smells, there is a disparity caused by the different metrics and features selection for the different techniques, given that the performance of the algorithm can only be as good as its input [11]. Some techniques provided metrics on the technique level, but did not give information on smell level and were excluded from comparison. In this work we found papers using search-based approach as GAs to improve the automation of refactoring, for this purpose of finding code smells. For instance, despite the GA is the technique that more appeared in the researches, but none of papers referenced their results separated by each code smell, difficultly comparisons.

This study also found that the papers lack comparable results, using the same data and performance metrics, a recurring problem in a significant number of studies so far [4, 22, 62, 63]. They also aim at the same smells, showing a high redundancy between the different techniques, the same happened when comparing tools as identified by Rasool and Arshad [62], Fernandes *et al.* [22]. This factors turn the

comparison of performance metrics between techniques a harder and inaccurate task, making the results less reliable and the studies harder to reproduce.

7. Threats to Validity

We have selected the search terms according to the research questions, taking into consideration the defined acceptance criteria and have used them to retrieve the relevant studies in the four electronic databases. Although some relevant studies may not use the terms related to the research questions in their titles, abstracts or keywords. We also left out broader terms such as refactoring and anti-patterns on purpose, in order to reduce the noise during the research stage, since terms can be referred to other fields out of code smells, leading to unrelated papers. As a result, we may have the high risk of leaving out these studies in the automatic search process. In order to mitigate this risk we have defined a selection criteria that strictly complies with the research questions to prevent the desired studies from being mistakenly excluded. In addition to that, the decision regarding study selection was made through double confirmation, taking separate selections by graduated researchers and a disagreement resolution for the divergent selections. However, relevant studies may have been missed. If such studies do exist, we believe that the number of them is reduced.

Another threat to validity is that the papers use different projects to assess the results, but even the studies that use the same project use different annotations to train the data, decreasing the reliability of the comparisons of performance. This threat is increased by publication bias as the researches tend to release only positive results, avoiding the negative ones, and also tend to show that their results outperform the others. In order to mitigate this threat we have registered the baseline that each study used, avoiding the comparison of studies developed on different projects with different baselines.

8. Conclusions

This study reviewed 26 papers covering machine learning techniques for code smells identification. For each of these smells, we evaluated the studied smells, the techniques and how they relate to each other and the performance of each of these techniques for the code smells.

We found out that the techniques perform close to each other, but Decision Tree, Random Forest, Semi-supervised and Nearest Neighbor techniques had better performance overall, besides the fact that they also tend to be heterogeneous covering smells from different types, due to this fact, the techniques tend to have a high redundancy, without specializing in a single smell. Exceptions for those findings were the Bayes approaches that performed worst than the others in general. The Association Rules and Decision Tree algorithms displayed better usage for smells that involve the relationship between different methods, classes and structures.

We also faced problems regarding the lack of comparability of the studies, since the studies use annotations done by their own personal instead of using a common base, they also did not publish the results using the same metrics, making it harder to compare and assess their performance, reducing the reliability of the study and performance assertions.

Acknowledgments

This work is partially funded by FAPEMIG Brazil.

Appendix A

Table A.1. F-Measure summary per smell and technique: Ordered by the median F-measure.

Smell	Technique	Min	Median	Max
Middle Man	Decision Tree	1.00	1.00	1.00
	Linear Statistics	0.85	0.85	0.85
	Nearest Neighbor	0.85	0.85	0.85
	Random Forest	0.71	0.71	0.71
	Naive Bayes Classifier	0.14	0.38	0.62
Speculative Generality	Association Rules (AR)	0.86	1.00	1.00
Divergent Change	Association Rules (AR)	0.55	0.85	1.00
Long Method	Association Rules (AR)	0.99	0.99	0.99
	Random Forest	0.86	0.92	0.99
	Decision Tree	0.86	0.92	0.98
	Support Vector Machine	0.69	0.83	0.97
	Naive Bayes Classifier	0.54	0.60	0.93
	Nearest Neighbor	0.89	0.90	0.90
	Linear Statistics	0.84	0.84	0.84
	Text-Based	0.56	0.62	0.71
Large Class	Random Forest	0.97	0.97	0.97
	Association Rules (AR)	0.97	0.97	0.97
	Decision Tree	0.96	0.96	0.96
	Naive Bayes Classifier	0.96	0.96	0.96
	Support Vector Machine	0.73	0.85	0.96
Long Parameter List	Random Forest	0.97	0.97	0.97
	Linear Statistics	0.95	0.95	0.95
	Nearest Neighbor	0.92	0.92	0.92
	Decision Tree	0.92	0.92	0.92
	Naive Bayes Classifier	0.31	0.32	0.33
Shotgun Surgery	Decision Tree	0.97	0.97	0.97
	Nearest Neighbor	0.89	0.90	0.91
	Linear Statistics	0.89	0.89	0.89
	Random Forest	0.89	0.89	0.89
	Naive Bayes Classifier	0.52	0.61	0.70
Feature Envy	Decision Tree	0.96	0.96	0.96
	Support Vector Machine	0.69	0.83	0.96
	Naive Bayes Classifier	0.24	0.26	0.95

Table A.1. (*Continued*)

Smell	Technique	Min	Median	Max
	Random Forest	0.94	0.94	0.94
	Association Rules (AR)	0.86	0.86	0.86
	Nearest Neighbor	0.84	0.84	0.84
	Linear Statistics	0.75	0.75	0.75
Duplicated Code	Semi-supervised	0.96	0.96	0.96
	Association Rules (AR)	0.72	0.85	0.85
Message Chains	Linear Statistics	0.86	0.86	0.86
	Nearest Neighbor	0.86	0.86	0.86
	Random Forest	0.80	0.80	0.80
	Naive Bayes Classifier	0.67	0.71	0.76
	Decision Tree	0.66	0.66	0.66
God Class	Clustering	0.66	0.72	0.82
BLOB	Bayesian Networks (BN)	0.60	0.69	0.79

Table A.2. Precision and recall summary per smell and technique: Ordered by the median precision.

		Precision			Recall		
Technique	Smell description	Min	Mean	Max	Min	Mean	Max
Decision Tree	Middle Man	1.00	1.00	1.00	1.00	1.00	1.00
Decision Tree	Shotgun Surgery	0.94	0.94	0.94	0.99	0.99	0.99
Decision Tree	Long Parameter List	0.86	0.86	0.86	0.98	0.98	0.98
Decision Tree	Long Method	0.77	0.77	0.77	0.97	0.97	0.97
Decision Tree	Message Chains	0.60	0.60	0.60	0.73	0.73	0.73
Semi-supervised	Duplicated Code	1.00	1.00	1.00	0.94	0.94	0.94
Association Rules (AR)	Speculative Generality	0.75	0.94	1.00	1.00	1.00	1.00
Association Rules (AR)	Duplicated Code	0.75	0.85	0.90	0.60	0.79	0.90
Association Rules (AR)	Divergent Change	0.50	0.80	1.00	0.50	0.76	1.00
Random Forest	Long Parameter List	0.95	0.95	0.95	0.99	0.99	0.99
Random Forest	Feature Envy	0.89	0.89	0.89	0.99	0.99	0.99
Random Forest	Shotgun Surgery	0.81	0.81	0.81	0.98	0.98	0.98
Random Forest	Long Method	0.77	0.77	0.77	0.96	0.96	0.96
Random Forest	Middle Man	0.76	0.76	0.76	0.67	0.67	0.67
Random Forest	Message Chains	0.75	0.75	0.75	0.87	0.87	0.87
Linear Statistics	Long Parameter List	0.92	0.92	0.92	0.99	0.99	0.99
Linear Statistics	Middle Man	0.88	0.88	0.88	0.83	0.83	0.83
Linear Statistics	Message Chains	0.86	0.86	0.86	0.87	0.87	0.87
Linear Statistics	Shotgun Surgery	0.81	0.81	0.81	0.98	0.98	0.98
Linear Statistics	Long Method	0.74	0.74	0.74	0.96	0.96	0.96
Linear Statistics	Feature Envy	0.62	0.62	0.62	0.97	0.97	0.97
Nearest Neighbor	Middle Man	0.88	0.88	0.88	0.83	0.83	0.83
Nearest Neighbor	Long Parameter List	0.87	0.87	0.87	0.98	0.98	0.98
Nearest Neighbor	Message Chains	0.86	0.86	0.86	0.87	0.87	0.87
Nearest Neighbor	Shotgun Surgery	0.81	0.83	0.85	0.98	0.98	0.98
Nearest Neighbor	Long Method	0.83	0.83	0.83	0.98	0.98	0.98
Nearest Neighbor	Feature Envy	0.73	0.73	0.73	0.98	0.98	0.98
Clustering	God Class	0.54	0.67	0.77	0.75	0.82	0.88
Naive Bayes Classifier	Message Chains	0.67	0.70	0.73	0.67	0.73	0.80
Naive Bayes Classifier	Middle Man	0.54	0.60	0.67	0.08	0.33	0.58

Table A.2. (*Continued*)

Technique	Smell description	Precision			Recall		
		Min	Mean	Max	Min	Mean	Max
Naive Bayes Classifier	Shotgun Surgery	0.38	0.47	0.56	0.83	0.88	0.92
Naive Bayes Classifier	Long Method	0.40	0.43	0.47	0.82	0.84	0.85
Naive Bayes Classifier	Long Parameter List	0.21	0.23	0.24	0.54	0.55	0.56
Naive Bayes Classifier	Feature Envy	0.14	0.15	0.16	0.73	0.73	0.74
Text-Based	Long Method	0.63	0.66	0.67	0.50	0.61	0.77
Bayesian Networks (BN)	BLOB	0.43	0.54	0.65	1.00	1.00	1.00

Table A.3. Articles selected for SLR.

Article	Author
A Bayesian Approach for the Detection of Code and Design Smells	[39]
An expert system for determining candidate software classes for refactoring	[43]
Automated scheduling for clone-based refactoring using a competent GA	[45]
Automatic Metric Thresholds Derivation for Code Smell Detection	[28]
Bad-smell prediction from software design model using machine learning techniques	[49]
BDTEX: A GQM-based Bayesian approach for the detection of anti-patterns	[40]
Co-changing code volume prediction through association rule mining and linear regression model	[46]
Code Bad Smell Detection through Evolutionary Data Mining	[30]
Code Smell Detection: Towards a Machine Learning-Based Approach	[27]
Code-Smell Detection As a Bilevel Problem	[64]
Evolution of Legacy System Comprehensibility Through Automated Refactoring	[34]
Hidden Truths in Dead Software Paths	[19]
High Dimensional Search-based Software Engineering	[53]
IDE-based Real-time Focused Search for Near-miss Clones	[75]
Identification and application of Extract Class refactorings in object-oriented systems	[24]
Identifying Extract Class refactoring opportunities using structural and semantic cohesion measures	[5]
Identifying Method Friendships to Remove the Feature Envy Bad Smell	[59]
Mining static and dynamic crosscutting concerns: a role-based approach	[7]
Mining Version Histories for Detecting Code Smells	[61]
Model refactoring using examples: a search-based approach	[31]
On the Use of Time Series and Search Based Software Engineering for Refactoring Recommendation	[70]
Search-based Determination of Refactorings for Improving the Class Structure of Object-oriented Systems	[65]
Software Analytics in Practice: A Defect Prediction Model Using Code Smells	[66]
Software Refactoring Under Uncertainty: A Robust Multi-objective Approach	[54]
Textual Analysis for Code Smell Detection	[60]
Using Concept Analysis to Detect Co-change Patterns	[32]

References

1. A. Abran and H. Nguyenkim, Measurement of the maintenance process from a demand-based perspective, *J. Softw. Maintenance: Res. Pract.* **5**(2) (1993) 63–90.
2. K. K. Aggarwal, Y. Singh and J. K. Chhabra, An integrated measure of software maintainability, in *Reliability and Maintainability Symp.*, 2002, pp. 235–241.

3. F. M. Akay, Support vector machines combined with feature selection for breast cancer diagnosis, *Expert Syst. Appl.* **36**(2) (2009) 3240–3247.
4. J. Al Dallal, Identifying refactoring opportunities in object-oriented code: A systematic literature review, *Inform. Softw. Technol.* **58** (2015) 231–249.
5. G. Bavota, A. De Lucia and R. Oliveto, Identifying extract class refactoring opportunities using structural and semantic cohesion measures, *J. Syst. Softw.* **84**(3) (2011) 397–414.
6. K. H. Bennett and V. T. V. T. Rajlich, Software maintenance and evolution: A roadmap, in *Proc. Conf. on the Future of Software Engineering*, 2000, pp. 73–87.
7. M. L. Bernardi, M. Cimitile and G. Di Lucca, Mining static and dynamic crosscutting concerns: A role-based approach, *J. Softw. Evolut. Process* **28**(5) (2016) 306–339.
8. W. J. Brown, R. C. Malveau, T. J. Mowbray and J. Wiley, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, Vol. 3 (John Wiley & Sons, 1998).
9. S. Bryton and F. B. Abreu, Strengthening refactoring: Towards software evolution with quantitative and experimental grounds, in *4th Int. Conf. Software Engineering Advances*, 2009, pp. 570–575.
10. S. Bryton, F. Brito, E. Abreu and M. Monteiro, Reducing subjectivity in code smells detection: Experimenting with the long method, in *Proc. 7th Int. Conf. on the Quality of Information and Communications Technology*, 2010, pp. 337–342.
11. C. Chakraborty and A. Joseph, Machine learning at central banks, Bank of England working papers 674, Bank of England, 2017, https://ideas.repec.org/p/boe/boeewp/0674.html.
12. A. Chatzigeorgiou and A. Manakos, Investigating the evolution of bad smells in object-oriented code, in *7th Int. Conf. on the Quality of Information and Communications Technology*, 2010, pp. 106–115.
13. A. Chatzigeorgiou and A. Manakos, Investigating the evolution of code smells in object-oriented systems, *Innov. Syst. Softw. Eng.* **10**(1) (2014) 3–18.
14. S. Counsell, R. M. Hierons, H. Hamza, S. Black and M. Durrand, Is a strategy for code smell assessment long overdue? in *Proc. of ICSE Workshop on Emerging Trends in Software Metrics*, 2010, pp. 32–38.
15. R. G. Cowell, R. J. Verrall and Y. K. Yoon, Modelling operational risk with Bayesian networks, *J. Risk Insur.* **74**(4) (2007) 795–827.
16. N. Dalkey and O. Helmer, An experimental application of the delphi method to the use of experts, *Manag. Sci.* **9**(3) (1963) 458–467.
17. A. Doostmohammadi, N. Amjady and H. Zareipour, Day-ahead financial loss/gain modeling and prediction for a generation company, *IEEE Trans. Power Syst.* **5**(3) (2017) 3360–3372.
18. T. Dybå and T. Dingsøyr, Strength of evidence in systematic reviews in software engineering, in *Proc. 2nd ACM-IEEE Int. Symp. on Empirical Software Engineering and Measurement*, 2008, pp. 178–187.
19. M. Eichberg, B. Hermann, M. Mezini and L. Glanz, Hidden truths in dead software paths, in *Proc. 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 474–484.
20. N. Fenton and M. Neil, Managing risk in the modern world, *Appl. Bayesian Netw.* **1**(1) (2007) 1–28.
21. V. Ferme, A. Marino and F. A. Fontana, Is it a real code smell to be removed or not? in *Int. Workshop on Refactoring & Testing*, 2013.
22. E. Fernandes, J. Oliveira, G. Vale, T. Paiva and E. Figueiredo, A review-based comparative study of bad smell detection tools, in *Proc. of 20th Int. Conf. on Evaluation and Assessment in Software Engineering*, 2016, pp. 1–12.

23. M. Fokaefs, N. Tsantalis and A. Chatzigeorgiou, Jdeodorant: Identification and removal of feature envy bad smells, *IEEE Int. Conf. on Software Maintenance*, 2007, pp. 519–520.
24. M. Fokaefs, N. Tsantalis, E. Stroulia and A. Chatzigeorgiou, Identification and application of extract class refactorings in object-oriented systems, *J. Syst. Softw.* **85**(10) (2012) 2241–2260.
25. F. Fontana, M. V. Mäntylä, M. Zanoni and A. Marino, Comparing and experimenting machine learning techniques for code smell detection, *Empir. Softw. Eng.* **21**(3) (2016) 1143–1191.
26. F. A. Fontana, P. Braione and M. Zanoni, Automatic detection of bad smells in code: An experimental assessment, *J. Object Technol.* **11**(2) (2012) 1–5.
27. F. A. Fontana, M. Zanoni, A. Marino and M. V. Mäntylä, Code smell detection: Towards a machine learning-based approach, in *IEEE Int. Conf. on Software Maintenance*, 2013, pp. 396–399.
28. F. A. Fontana, V. Ferme, M. Zanoni and A. Yamashita, Automatic metric thresholds derivation for code smell detection, in *Int. Workshop on Emerging Trends in Software Metrics*, 2015, pp. 44–53.
29. M. Fowler and K. Beck, *Refactoring: Improving the Design of Existing Code* (Addison-Wesley, 1999).
30. S. Fu and B. Shen, Code bad smell detection through evolutionary data mining, in *Int. Symp. on Empirical Software Engineering and Measurement*, 2015, pp. 41–49.
31. A. Ghannem, G. El Boussaidi and M. Kessentini, Model refactoring using examples: A search-based approach, *J. Softw.: Evolut. Process* **26**(7) (2014) 692–713.
32. T. Gîrba, S. Ducasse, A. Kuhn, R. Marinescu and R. Daniel, Using concept analysis to detect co-change patterns, in *9th Int. Workshop on Principles of Software Evolution: In Conjunction with 6th ESEC/FSE Joint Meeting*, Vol. 6, 2007, p. 89.
33. D. E. Goldberg and J. H. Holland, Genetic algorithms and machine learning, *Mach. Learn.* **3**(2) (1988) 95–99.
34. I. Griffith, S. Wahl and C. Izurieta, Evolution of legacy system comprehensibility through automated refactoring, in *Proc. Int. Workshop on Machine Learning Technologies in Software Engineering*, 2011, pp. 35–42.
35. M. Hozano, A. Garcia, B. Fonseca and E. Costa, Are you smelling it? investigationg how similar developers detect code smells, *Inform. Softw. Technol.* **93**(1) (2017) 130–146.
36. A. Jain, Data clustering: 50 years beyond K-meansstar, open, *Pattern Recogn. Lett.* **31**(8) (2010) 651–666.
37. A. K. Jain, M. N. Murty and P. J. Flynn, Data clustering: A review, *ACM Comput. Surv.* **31**(3) (1999) 264–323.
38. D. B. Kell, Metabolomics, machine learning and modelling: Towards an understanding of the language of cells, *Biochem. Soc. Trans.* **33**(Pt 3) (2005) 520–524.
39. F. Khomh, S. Vaucher, Y. G. Gueheneuc and H. Sahraoui, A Bayesian approach for the detection of code and design smells, in *9th Int. Conf. on Quality Software*, 2009, pp. 305–314.
40. F. Khomh, S. Vaucher, Y. G. Guéhéneuc and H. Sahraoui, BDTEX: A GQM-based Bayesian approach for the detection of antipatterns, *J. Syst. Softw.* **84**(4) (2011) 559–572.
41. B. Kitchenham, R. Pretorius, D. Budgen, O. P. Brereton, M. Turner, M. Niazi and S. Linkman, Systematic literature reviews in software engineering — A tertiary study, *Inform. Softw. Technol.* **52**(8) (2010) 792–805.
42. B. A. Kitchenham, D. Budgen and P. Brereton, *Evidence-Based Software Engineering and Systematic Reviews* (CRC Press, 2015).
43. Y. Kosker, B. Turhan and A. Bener, An expert system for determining candidate software classes for refactoring, *Expert Syst. Appl.* **36**(6) (2009) 10000–10003.

44. S. B. Kotsiantis, Supervised machine learning: A review of classification techniques, *Inform. An Int. J. Comput. Inform.* **3176**(31) (2007) 249–268.
45. S. Lee, G. Bae, H. S. Chae, D. H. Bae and Y. R. Kwon, Automated scheduling for clone-based refactoring using a competent GA, *Softw. Practice Exper.* **41**(5) (2011) 521–550.
46. S. J. Lee, L. H. Lo, Y. C. Chen and S. M. Shen, Co-changing code volume prediction through association rule mining and linear regression model, *Expert Syst. Appl.* **45** (2016) 185–194.
47. Y. Li, Deep reinforcement learning: An overview, arXiv:1701.07274.
48. H. Liu, Z. Ma, W. Shao and Z. Niu, Schedule of bad smell detection and resolution: A new way to save effort, *IEEE Trans. Softw. Eng.* **38**(1) (2012) 220–235.
49. N. Maneerat and P. Muenchaisri, Bad-smell prediction from software design model using machine learning techniques, in *8th Int. Joint Conf. on Computer Science and Software Engineering*, 2011, pp. 331–336.
50. M. Mantyla, J. Vanhanen and C. Lassenius, A taxonomy and an initial empirical study of bad smells in code, *Int. Conf. on Software Maintenance*, 2003, pp. 381–384.
51. M. Mantyla, J. Vanhanen and C. Lassenius, Bad smells: Humans as code critics, in *20th IEEE Int. Conf. on Software Maintence*, 2004, pp. 399–408.
52. R. Marinescu, Detection strategies: Metrics-based rules for detecting design flaws, in *IEEE Int. Conf. on Software Maintenance*, 2004, pp. 350–359.
53. M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb and M. Ó'Cinnéide, High dimensional search-based software engineering, in *Proc. Conf. on Genetic and Evolutionary Computation*, 2014, pp. 1263–1270.
54. M. W. Mkaouer, M. Kessentini, S. Bechikh, M. Ó'Cinnéide and K. Deb, Software refactoring under uncertainty, in *Proc. Conf. Companion on Genetic and Evolutionary Computation Companion*, 2014, pp. 187–188.
55. N. Moha and Y. Guéhéneuc, DECOR: A method for the specification and detection of code and design smells, *IEEE Trans. Softw. Eng.* **36**(1) (2010) 20–36.
56. E. Murphy-hill, C. Parnin and A. P. Black, How we refactor, and how we know it, *IEEE Trans. Softw. Eng.* **38**(1) (2012) 55–57.
57. D. R. Newman, The use of linkage learning in genetic algorithms, 2006.
58. S. Olbrich, D. S. Cruzes, V. Basili and N. Zazworka, The evolution and impact of code smells: A case study of two open source systems what are code smells? in *Proc. 3rd Int. Symp. Empirical Software Engineering and Measurement*, 2009, pp. 390–400.
59. R. Oliveto, M. Gethers, G. Bavota, D. Poshyvanyk and A. De Lucia, Identifying method friendships to remove the feature envy bad smell, in *Proc. 33rd Int. Conf. Software Eng.*, 2011, p. 820.
60. Palomba and Fabio, Textual analysis for code smell detection, in *Proc. 37th Int. Conf. on Software Engineering*, Vol. 2, 2015, pp. 769–771.
61. F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk and A. De Lucia, Mining version histories for detecting code smells, *IEEE Trans. Softw. Eng.* **41**(5) (2015) 462–489.
62. G. Rasool and Z. Arshad, A review of code smell mining techniques, *J. Softw.: Evolut. Process* **27**(11) (2015) 867–895.
63. D. Rattan, R. Bhatia and M. Singh, Software clone detection: A systematic review, *Inform. Softw. Technol.* **55**(7) (2013) 1165–1199.
64. D. Sahin, M. Kessentini, S. Bechikh and K. Deb, Code-smell detection as a bilevel problem, *ACM Trans. Softw. Eng. Methodol.* **24**(1) (2014) 1–44.
65. O. Seng, J. Stammel and D. Burkhart, Search-based determination of refactorings for improving the class structure of object-oriented systems, in *Proc. 8th Annual Conf. on Genetic and Evolutionary Computation*, 2006, pp. 1909–1916.

66. B. Soltanifar, S. Akbarinasaji, B. Caglayan, A. B. Bener, A. Filiz and B. M. Kramer, Software analytics in practice, in *Proc. 20th Int. Symp. on Database Engineering & Applications*, 2016, pp. 148–155.
67. D. Taibi, A. Janes and V. Lenarduzzi, How developers perceive smells in source code: A replicated study, *Inform. Softw. Technol.* **92**(Supplement C) (2017) 223–235.
68. M. Tufano, F. Palomba, G. Bavota, R. Olivetox, M. Di Penta, A. De Lucia and D. Poshyvanyk, When and why your code starts to smell bad, in *Proc. Int. Conf. on Software Engineering*, Vol. 1, 2015, pp. 403–414.
69. B. Walter and T. Alkhaeir, The relationship between design patterns and code smells: An exploratory study, *Inform. Softw. Technol.* **74** (2016) 127–142.
70. H. Wang, M. Kessentini, W. Grosky and H. Meddeb, On the use of time series and search based software engineering for refactoring recommendation, in *Proc. 7th Int. Conf. on Management of Computational and Collective Intelligence in Digital EcoSystems*, Vol. 7, 2015, pp. 35–42.
71. J. Wen, S. Li, Z. Lin, Y. Hu and C. Huang, Systematic literature review of machine learning based software development effort estimation models, *Inform. Softw. Technol.* **54**(1) (2012) 41–59.
72. H. Zhang, M. A. Babar and P. Tell, Identifying relevant studies in software engineering, *Inform. Softw. Technol.* **53**(6) (2011) 625–637.
73. M. Zhang, T. Hall and N. Baddoo, Code Bad Smells: A review of current knowledge, *J. Softw. Mainten. Evolut.* **23**(3) (2011) 179–202.
74. X. Zhu, Semi-supervised learning, in *Encyclopedia of Machine Learning* (Springer, 2011), pp. 892–897.
75. M. F. Zibran and C. K. Roy, IDE-based real-time focused search for near-miss clones, in *Proc. 27th Annual ACM Symp. on Applied Computing*, 2012, pp. 1235–1242.