Machine Learning Techniques for Code Smell Detection: A Systematic Literature Review and Meta-Analysis

Muhammad Ilyas Azeem[a,b], Fabio Palomba[d], Lin Shi[a,b], Qing Wang[a,b,c]

[a]Laboratory for Internet Software Technologies, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China.
[b]University of Chinese Academy of Sciences, Beijing 100049, China.
[c]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China.
[d]University of Zurich, Zurich, Switzerland.

Abstract

Background: Code smells indicate suboptimal design or implementation choices in the source code that often lead it to be more change- and fault-prone. Researchers defined dozens of code smell detectors, which exploit different sources of information to support developers when diagnosing design flaws. Despite their good accuracy, previous work pointed out three important limitations that might preclude the use of code smell detectors in practice: (i) subjectiveness of developers with respect to code smells detected by such tools, (ii) scarce agreement between different detectors, and (iii) difficulties in finding good thresholds to be used for detection. To overcome these limitations, the use of machine learning techniques represents an ever increasing research area.

Objective: While the research community carefully studied the methodologies applied by researchers when defining heuristic-based code smell detectors, there is still a noticeable lack of knowledge on how machine learning approaches have been adopted for code smell detection and whether there are points of improvement to allow a better detection of code smells. Our goal is to provide an overview and discuss the usage of machine learning approaches in the field of code smells.

Email addresses: azeem@itechs.iscas.ac.cn (Muhammad Ilyas Azeem), palomba@ifi.uzh.ch (Fabio Palomba), shilin@itechs.iscas.ac.cn (Lin Shi), wq@itechs.iscas.ac.cn (Qing Wang)

Method: This paper presents a Systematic Literature Review (SLR) on Machine Learning Techniques for Code Smell Detection. Our work considers papers published between 2000 and 2017. Starting from an initial set of 2,456 papers, we found that 15 of them actually adopted machine learning approaches. We studied them under four different perspectives: (i) code smells considered, (ii) setup of machine learning approaches, (iii) design of the evaluation strategies, and (iv) a meta-analysis on the performance achieved by the models proposed so far.

Results: The analyses performed show that *God Class*, *Long Method*, *Functional Decomposition*, and *Spaghetti Code* have been heavily considered in the literature. Decision Trees and Support Vector Machines are the most commonly used machine learning algorithms for code smell detection. Models based on a large set of independent variables have performed well. JRip and Random Forest are the most effective classifiers in terms of performance. The analyses also reveal the existence of several open issues and challenges that the research community should focus on in the future.

Conclusion: Based on our findings, we argue that there is still room for the improvement of machine learning techniques in the context of code smell detection. The open issues emerged in this study can represent the input for researchers interested in developing more powerful techniques.

Keywords: Code Smells, Machine Learning, Systematic Literature Review.

---

1. Introduction

During software maintenance and evolution, software systems need to be continuously changed by developers in order to (i) implement new requirements, (ii) enhance existing features, or (iii) fix important bugs [1]. Due to time pressure or community-related factors [2], developers do not always have the time or the willingness to keep the complexity of the system under control and find good design solutions before applying their modifications [3]. As a consequence, the development activities are often performed in an undisciplined manner, and have the effect to erode the original design of the system by introducing the so-called *technical debt* [4, 5].

Code smells [6], i.e., symptoms of the presence of poor design or implementation choices in the source code, represent one of the most serious forms of technical debt [7, 8]. Indeed, previous research found that not only they strongly reduce the ability of developers to comprehend the source

code [9], but also make the affected classes more change- and fault-prone [10, 11, 12, 13]. Thus, they represent an important threat for maintainability effort and costs [14, 15, 16]. In past and recent years, the research community was highly active on the topic. On the one hand, many empirical studies have been conducted with the aim of understanding (i) when and why code smells are introduced [17, 18, 19], (ii) what is their evolution and longevity in software projects [20, 21, 22, 23, 24, 25, 26, 27, 28, 29], and (iii) to what extent they are relevant for developers [30, 31, 32, 33, 34, 35, 36, 37].

On the other hand, several code smell detectors have been proposed as well [38, 39, 40, 41, 42, 43, 44, 45, 46, 47]. Most of them can be considered as *heuristics-based*: they apply a two-step process where a set of metrics are firstly computed, and then some thresholds are applied upon such metrics to discriminate between smelly and non-smelly classes. They differ from each other for (i) the specific algorithms used to identify code smells (e.g., a combination of metrics or through the use of more advanced methodologies like Relational Topic Modeling) and (ii) the metrics exploited (e.g., based on code metrics or historical data). Although it has been showed that such detectors have reasonable performance in terms of accuracy of the recommendations, previous works highlighted a number of important limitations that might preclude the use of such detectors in practice [48, 49]. In particular, code smells identified by existing detectors can be subjectively interpreted by developers [50, 51]. At the same time, the agreement between them is low [52]. More importantly, most of them require the specification of thresholds to distinguish smelly code components from non-smelly ones [48]: naturally, the selection of thresholds strongly influence their accuracy.

For all these reasons, a recent trend is the adoption of Machine Learning (ML) techniques for approaching the problem [53]. In this scenario, a supervised method is exploited: a set of independent variables (a.k.a., *predictors*) are used to predict the value of a dependent variable (i.e., the smelliness of a class) using a machine learning classifier (e.g., Logistic Regression [54]). The model can be trained using a sufficiently large amount of data available from the project under analysis, i.e., *within-project* strategy, or using data coming from other (similar) software projects, i.e., *cross-project* strategy. These approaches clearly differ from the heuristics-based ones, as they rely on classifiers to discriminate the smelliness of classes rather than on predefined thresholds upon computed metrics.

While the research community heavily studied the methodologies adopted by researchers and practitioners in the context of heuristics-based code smell

detectors [48, 52, 49], only a little knowledge is available on the strategies applied for building code smell prediction models. We believe that this piece of information is extremely important for researchers interested in devising novel effective methodologies to deal with code smells.

To cope with this lack of knowledge, in the paper we conducted a Systematic Literature Review (SLR) on the usage of ML techniques for code smell detection—covering the papers published between 2000 and 2017—with the aim of (i) *understanding and summarizing* the current state of the art in this field and (ii) *analyzing its limitations and open challenges* in order to drive future research.

More specifically, our SLR aims at providing a comprehensive investigation to elaborate (i) the types of code smells taken into account by previous research, (ii) the dependent and independent variables proposed in literature to identify code smells, (iii) the types of classifiers exploited by researchers, and (iv) the training strategies used to train and evaluate the machine learning techniques. Moreover, we report a meta-analysis of the performance of the machine learning models defined so far. To this aim, we set up the research questions reported in Table 1.

Besides the mere analysis of the state of the art, we also aim at discovering and reporting the limitations of the current approaches in terms of classifier selection, dependent and independent variables, training strategy, and the evaluation approaches adopted so far.

1.1. Related Research

To the best of our knowledge, no Systematic Literature Review has been conducted with the aim of understanding and summarizing the research on code smell prediction models. However, it is important to point out that some secondary studies on code smells and code smell detection tools have been proposed [48, 49, 55, 56, 57].

Specifically, the SLR conducted by Zhang et al. [49] covered the state-of-the-art on code smells between 2000 and June 2009 with a focus on (i) which code smells have received more attention, (ii) what design methodologies have been used to study the phenomenon, and (iii) what was the goal of such studies (e.g., empirical observations or devising of new techniques). As an additional analysis, they also explored whether any published study provided empirical evidence to support the negative effects of code smells [6].

Wangberg and Yamashita [55] performed another SLR on code smells and refactoring. The aim of the study was to get an overview of research related

Table 1: Research Questions posed for our Systematic Literature Review

Research Question	Motivation
RQ1 - Code Smells Considered	
• **RQ1**: Which code smells can be currently detected using machine learning techniques?	To explore the current state-of-the-art of the code smell detection using machine learning techniques with respect to the code smells considered so far.
RQ2 - Machine Learning Setup	
• **RQ2.1**: What independent variables have been considered to predict the presence of code smells? • **RQ2.2**: What classification types have been considered to identify the smelliness of source code artifacts? • **RQ2.3**: What machine learning algorithms have been used for code smell identification? • **RQ2.4**: What training strategies have been proposed in the literature?	To analyze the machine learning settings adopted by previous research with respect to independent and dependent variables, machine learning algorithms, and training methodologies. Answers to these questions will help both the practitioners and researches to select the best machine learning setup for code smell detection: Which independent variables (reduces feature engineering efforts required to select the best feature from the corpus) should be used with which machine learning algorithm(s) using which training strategy to produce best prediction results.
RQ3 - Evaluation Setup	
• **RQ3.1**: What types of validation techniques have been exploited? • **RQ3.2**: Which have been the evaluation metrics used to access code smell prediction models? • **RQ3.3**: Which have been the datasets considered?	To study the methodologies exploited to (i) validate the proposed code smell prediction models, (ii) evaluate their accuracy, and (iii) analyze the source code projects taken into account by researchers.
RQ4 - Performance Meta-Analysis	
• **RQ4.1**: Which independent variables were reported to perform better for code smell prediction? • **RQ4.2**: Does the machine learning algorithm impact the performance of code smell prediction models? • **RQ4.3**: Does the training strategy impact the performance of code smell prediction models?	To study the extent to which independent variables, machine learning approaches, and training strategies impact the performance of code smell prediction models.

to each of the stages of the refactoring process: (i) detecting code smells, (ii) making decisions on which refactoring to choose, and (iii) performing the

refactoring. The review also attempted to identify which methods and tools have been created to support these various stages of refactoring.

The SLR proposed by Vale et al. [56] investigated the phenomenon of code smells in the context of Software Product Lines (SPL). The study had the goal to identify SPL-specific code smells and refactoring methods. Indeed, as a result, they came up with a catalog of code smells and refactoring methods for Software Product Lines. Fernandes et al. [48] performed an SLR on code smells detection tools, targeting papers from 2000 to 2016. The study aimed at identifying all the code smells detection tools, their key features and code smell types they are able to identify. Besides, a comparison of the four most frequently occurred tools was made based on recall, precision, usability and agreement among the tools. Finally, Rasool and Arshad [57] also conducted an SLR on techniques and tools used for mining code smells from the source code. State-of-the-art tools and techniques were classified based on the detection methods and their results were analyzed. They proposed some recommendations about the tools for developers working in the field of code smell detection.

With respect to the papers discussed above, it is important to point out that none of them explicitly targeted machine learning approaches for code smell detection. Fernandes et al. [48] focused on tools rather than techniques for code smell detection: as no tool implements a machine learning model, the authors did not include ML-based approaches in their review. Vale et al. [56] took into account code smells in the context of product lines, highlighting which smells exist in that context and what are the techniques that can be applied: since no machine learning approach has been defined for that purpose, the authors did not overview ML-based methods. Rasool and Arshad [57] focused on mining approaches, and therefore they did not take into account machine learning models. The papers by Zhang et al. [49] and Wangberg and Yamashita [55] focused instead on the research on code smells and refactoring: as such, they included both empirical studies and approaches enabling the entire refactoring process. In their works, some machine learning approaches identified as primary studies in our work (i.e., the [S01], [S03], [S13] presented later in the paper) were cited; however, the authors did not focus on the characteristics of such techniques but limited to signal their presence. Thus, they did not analyze the specific machine learning settings adopted by researchers, the independent and dependent variables considered, and the validation techniques exploited to evaluate the performance of code smell prediction models.

Since the usage of machine learning for code smell detection is highly promising [53], and given the proved impact of machine learning settings on the overall performance of prediction models [58, 59, 60, 61], we believe that a dedicated analysis is required in order to build additional knowledge on the topic and extract the open challenges that future research should focus on.

1.2. Contributions

The contributions made by this SLR are the following:

1. We identify a set of 15 primary studies that proposed code smell prediction models. The research community can use them as a starting point to extend the knowledge on the topic.

2. We present a comprehensive synthesis of the primary studies identified. This includes four main themes: (i) code smell considered, (ii) setup of machine learning approaches, (iii) design of the evaluation strategies, and (iv) performance analysis of the proposed models.

3. We provide guidelines and recommendations based on our findings to support further research in the area.

4. We provide a comprehensive replication package[1] containing all the data and analysis scripts used to conduct this SLR.

2. Research Methodology

A systematic literature review has been used as a research methodology in this study as it is a defined and methodical way of identifying, assessing, and analyzing published literature in order to investigate a specific research question or a phenomenon [62]. As done by other researchers in the field of software engineering [63, 64, 65, 66], we adopted the SLR guidelines proposed by Kitchenham and Charters [62]. Furthermore, we integrated the procedure adopting the systematic inclusion of references, also known as "snowballing", defined by Wohlin [67]. The following subsections describe the process followed.

[1] https://figshare.com/articles/Replication_package/7370423

2.1. Search Strategy

We devised a search strategy to collect all the available published literature relevant to our topic. Our search strategy comprised of search terms identification, resources to be searched, search process and article selection criteria adopted for the studies.

2.1.1. Identifying search terms

To find the relevant search terms we followed five steps [62]:

a We used the research questions for the derivation of major terms, by identifying population, intervention, and outcome;

b For all the major terms, we found the alternative spellings and/or synonyms;

c We verified the keywords in any relevant paper;

d We used boolean operators for conjunction in case a certain database allows it, i.e., we used the **OR** operator for the concatenation of alternative spellings and synonyms whereas the **AND** operator for the concatenation of major terms;

e We integrated the search string into a summarized form if required.

Results for a). As for the first step, we identified population, intervention, and outcome in order to better design our search terms. Specifically:

- **Population**: Code smell detectors;
- **Intervention**: Machine Learning techniques;
- **Outcomes of relevance**: Code smells.

For instance, a research question containing the above details is:

> **RQ1.1**: Which [code smells] [**OUTCOMES**]
> can be currently [detected] [**POPULATION**]
> by using [machine learning techniques]? [**INTERVENTION**]

Results for b). The alternative spellings and synonyms identified are:

- **Code Smells**: ("code smells" OR "code smell" OR "code bad smells" OR "bad code smells" OR "bad smells" OR "anomalies" OR "anti-patterns" OR "antipattern" OR "design defect" OR "design-smells" OR "design flaw");
- **Machine Learning**: ("machine learning" OR "supervised learning" OR "classification" OR "regression" OR "unsupervised learning");
- **Prediction**: ("prediction" OR "detection" OR "identification" OR "prediction model" OR "model");
- **Software**: ("software" OR "software engineering").

Results for c). We checked the keywords in the relevant papers, and we did not find any other alternative spelling or synonym to add in the set of relevant terms to consider.

Results for d). We used boolean operators, coming up with the search query reported below:

(("code smells" OR "code smell" OR "code bad smells" OR "bad code smells" OR "bad smells" OR anomalies OR anti-patterns OR antipattern OR "design defect" OR "design-smells" OR "design flaw") AND ("machine learning" OR "supervised learning" OR classification OR regression OR "unsupervised learning") AND (software OR "software engineering"))

Results for e). Due to the search term limitation of the IEEE Xplore digital library, we also defined the short search string reported below:

(("code smells" OR "code bad smells" OR "bad smells" OR anti-patterns OR "design defect" OR "design-smells" OR "design flaw") AND ("machine learning" OR "supervised learning" OR "unsupervised learning") AND (detection OR identification OR "prediction model") AND (software OR "software engineering"))

2.1.2. Resources to be searched

Selection of proper resources to search for relevant literature plays a significant role in an SLR. We selected the following resources to search for all the available literature relevant to our research questions:

- IEEE Xplore digital library (http://ieeexplore.ieee.org)
- ACM digital library (https://dl.acm.org)
- ScienceDirect (http://www.sciencedirect.com)
- SpringerLink (https://link.springer.com)
- Scopus (https://www.scopus.com)
- Engineering Village (https://www.engineeringvillage.com)

The selection of these databases was driven by our willingness of gathering as many papers as possible to properly conduct our systematic literature review. In this respect, the selected sources are recognized as the most representative for Software Engineering research and are used in many other SLRs [62] because they contain a massive amount of literature, i.e. journal articles, conference proceedings, books etc., related to our research questions.

2.2. Article Selection Process

The article selection process followed in this study is depicted in Figure 1. The following subsections report details on the selection process.

2.2.1. Search process overview

To search all the available published papers relevant to our research questions, we followed four main steps:

a We used the search strings mentioned in Section 2.1.1 to collect the primary studies present in the digital libraries mentioned in Section 2.1.2. We did not put any date restriction on the search process to collect as much relevant literature as possible. The search results produced by the digital libraries is shown in the second column of Table 2: as shown, we found 2,456 papers respecting the query;

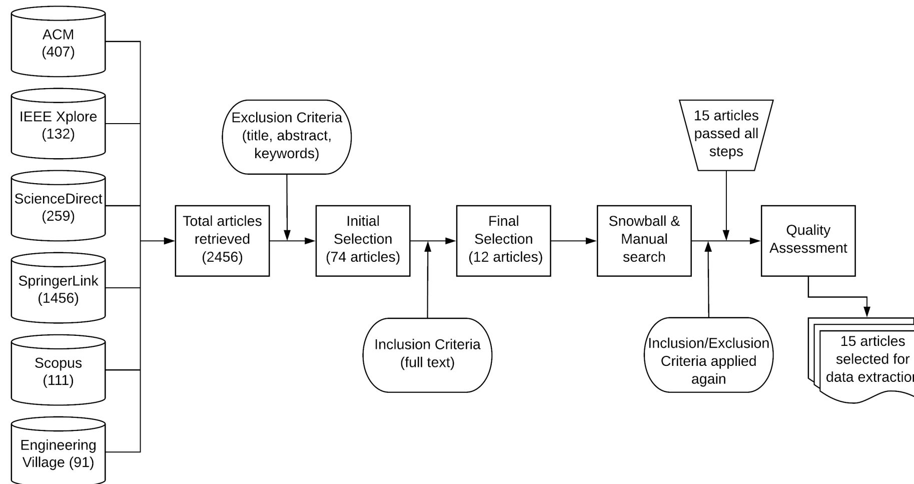

Figure 1: Article selection process

b Starting from the entire list of retrieved sources, we firstly filtered out non-relevant papers by applying the exclusion criteria (detailedly reported in Section 2.2.2). This step was conducted by the first author of this paper, who read the title, abstract and keywords of the 2,456 found papers and applied the exclusion criteria. The articles that satisfied such criteria were discarded, while the remaining ones were selected for the second stage of screening, i.e., the inclusion criteria. This process leads to our final selection for the search string, composed of 12 articles ($\approx 0.13\%$ of the papers found in the previous step);

c We performed a snowballing process to search for possible missing papers [67]. In particular, snowballing refers to the use of the reference list of a paper or the citations to the paper to identify additional sources [67]. In our context, we carried out both forward and backward snowballing: with the former, we included all the papers referenced in the initially selected papers, while with the latter we included all the papers that reference the initially selected papers. Afterward, the first author of this paper carried out the same process as the previous point, i.e., he read the title, abstract, and keywords of the 35 snowballed papers and re-applied the exclusion/inclusion criteria: as a result, only 2 papers (see Table 2) were found to be further analyzed;

Table 2: Data sources and search results.

Resource name	**Total results found**	**Initial Selection**	**Final Selection**
IEEE Xplore Digital Library	132	28	9
ACM Digital Library	407	11	1
ScienceDirect Digital Library	259	3	2
SpringerLink Digital Library	1456	16	0
Scopus Digital Library	111	7	0
Engineering Village Digital Library	91	9	0
Total	2456	74	12
Snowballing Process	35	9	2
Manual Search	≈13,300	35	1
Total	≈13,335	44	3
Grand Total	≈15,751	118	15

d To ensure that all the relevant literature was included in the final list of selected sources, we also performed an additional manual search covering all the papers published in the last ten years in the 21 most relevant software engineering conferences and 11 journals (the complete list of conferences and journals analyzed in this phase is given in Appendix A). More specifically, the first author of the paper scanned each of the ≈13,300 additional sources identified and based on the exclusion/inclusion criteria he decided whether to include it or not in the final selection. To this aim, he first considered the title: if this was clearly out of scope (e.g., if it referred to automatic test case generation), the paper was skipped; conversely, if a certain publication was considered potentially useful, the first author went through the abstract, introduction, and conclusion to further verify how appropriate it was for our study. The entire process took three work-weeks and, in the end, only 1 publication relevant to our research question was included;

e Given the set of sources finally discovered via search string and then augmented with those retrieved using snowballing and manual search, i.e., a set composed of 15 sources. Thus, overall 15 papers passed all the steps;

f Once we had established the set of final papers to consider in the SLR, we applied the last filtering, i.e., the quality assessment phase, to ensure that all the final papers had the information needed to answer our research questions (see Section 2.2.3). This step led to the final set of papers analyzed in our study. Of the 15 sources discovered until this step, all of them passed the quality assessment. Thus, our SLR is based on such 15 papers. The data extraction process is reported in Section 2.2.4.

It is important to note that the process described above was completely double-checked by the second author of this paper to ensure the accuracy of the selection process. To measure the agreement between the two inspectors in the inclusion/exclusion process, we computed an inter-rater reliability index: specifically, we measured the widely known Krippendorff's alpha Kr_α [68]. We found it to be 0.98, considerably higher than the 0.80 standard reference score [69] for Kr_α. As for the disagreements, the two authors opened a discussion in order to reach a consensus: as a result, the 15 papers previously selected as final sources were confirmed as such. The next subsections overview (i) inclusion/exclusion criteria, (ii) quality assessment process of the procedure reported above, and (iii) data extraction process.

2.2.2. Inclusion and Exclusion Criteria

To be useful for answering our research questions, a paper passed the following criteria.

a **Exclusion criteria:** Sources that met the constraints reported below were *excluded* from our study:

- Articles that were focused on other detection techniques rather than machine learning. This led to the exclusion of 1,958 papers;
- Articles that were not written in English. A set of 13 papers were excluded;

- Articles whose full text is not available. Further 411 papers were excluded.

b **Inclusion criteria:** Sources that met the constraints reported below were *included* in our study:

- All the articles, written in English, reporting machine learning techniques for code smells detection;
- Articles which introduce new techniques to improve the performance of the existing machine learning techniques used for code smell detection.

It is worth noting that we included all types of papers (i.e., journal, conference, workshop, and short papers) with the aim of collecting a set of relevant sources as more comprehensive as possible.

2.2.3. Study quality assessment

The quality of publications was measured after the final selection process. The following checklist was used to assess the credibility and thoroughness of the selected publications.

- **Q1**: *Are the code smells detected by the proposed technique clearly defined?*
- **Q2**: *Are the independent and dependent variables clearly defined?*
- **Q3**: *Is the machine learner classifier clearly defined?*
- **Q4**: *Are the evaluation strategies and metrics explicitly reported?*

Each of the above questions was marked as “Yes”, “Partially” or “No”. We considered a study as partial in cases where the methodological details could have been derived from the text, even if they were not clearly reported. As an example, consider the case in which a study reports precision and recall as evaluation metrics: details on the F-Measure of the proposed model could be simply derived starting from the values of precision and recall, even though the F-Measure was not explicitly reported. These answers were scored as follows: “Yes”=1, “Partially”=0.5, and “No”=0. For each selected

Table 3: Data extraction Form.

Dimension	Attribute: Description
Type of Code Smells studied	What type of code smells are detected by the proposed system?
Programming Language	The technique is applicable to which programming language e.g. Java, C++ etc.
Machine Learning algorithm used	What type of machine learning algorithm(s) has/have been used in the study?
Independent Variables	The predictors used to measure the proneness of a code component to be affected by a code smell e.g. LCOM, WMC etc.
Dependent Variables	The smelliness of a source code component like class level or method level etc.
Classification Type	What type of classification type has been used to report the code smell detection result e.g. binary classification or multi-valued classification etc?
Training strategy	What type of strategy i.e. within- or cross-project has been used to train the model?
Validation Techniques	What technique has been used to validate the model? E.g. k-fold cross-validation etc.
Evaluation Metrics	What evaluation metrics have been used to assess the accuracy of the model e.g. Precision, Recall, F Measure, and AUC-ROC etc?
Evaluation Methodology	**Preliminary**: The evaluation is on small systems or small datasets or only preliminary evidence is given (i.e., proof of concept) **Benchmark**: The evaluation uses a dataset that was published by other authors or the dataset used in this evaluation is later used by other researchers **Human subjects**: • **Academic**: Students or non-professional developers participated in the evaluation • **Professional**: Professional developers participated in the evaluation of the results **Qualitative**: The paper discusses details about the characteristics of the technique/tool and/or some aspects of the results **Comparison with other approaches**: Comparisons of the authors approach with existing solutions **Unknown/none**: There is no evaluation performed, or the details are not available
Dataset	What data set has been used to train the model? The list of the software projects upon which code smell detection techniques have been applied.
Limitations	The limitation of the existing technique

primary study, its quality score was computed by summing up the scores of the answers to all the four questions.

Table 4: The Reviewed Primary Studies

Ref.	Title	Author	Year	Publication Type
[S01]	Code Smell Detection: Towards a Machine Learning-Based Approach	Arcelli Fontana et al.	2013	Conference
[S02]	Tracking Design Smells: Lessons from a Study of God Classes	Vaucher et al.	2009	Conference
[S03]	A Bayesian Approach for the Detection of Code and Design Smells	Khomh et al.	2009	Conference
[S04]	IDS: An Immune-Inspired Approach for the Detection of Software Design Smells	Hassaine et al.	2010	Conference
[S05]	Bad-smell prediction from software design model using machine learning techniques	Maneerat and Muenchaisri	2011	Conference
[S06]	SMURF: A SVM-based Incremental Anti-pattern Detection Approach	Maiga et al.	2012	Conference
[S07]	Can I clone this piece of code here?	Wang et al.	2012	Conference
[S08]	Support vector machines for anti-pattern detection	Maiga et al.	2012	Conference
[S09]	Experience report: Evaluating the effectiveness of decision trees for detecting code smells	Amorim et al.	2015	Conference
[S10]	Code smell severity classification using machine learning techniques	Arcelli Fontana and Zanoni	2017	Journal
[S11]	Adaptive Detection of Design Flaws	Kreimer	2005	Journal
[S12]	BDTEX: A GQM-based Bayesian approach for the detection of antipatterns	Khomh et al.	2011	Journal
[S13]	Comparing and experimenting machine learning techniques for code smell detection	Arcelli Fontana et al.	2016	Journal
[S14]	Deep learning code fragments for code clone detection	White et al.	2016	Conference
[S15]	Classification model for code clones based on machine learning	Yang et al.	2015	Journal

The process was performed by the first two authors of this paper, who jointly evaluated each source. We classified the quality level into High (score = 4), Medium ($2 \leq$ score < 4), and Low (score < 2). We selected 15 studies that scored in high and medium levels as our final selection.

2.2.4. Data extraction

Once we had selected the final papers to be used for the SLR, we proceeded with the extraction of the data needed to answer our research questions. Specifically, we relied on the data extraction form presented in Table 3. Besides the information about the specific attributes under consideration, e.g., the code smell types considered in the selected papers, we also reserved a field aimed at reporting possible limitations of the considered studies. This eased the process of finding limitations of existing work as well as guidelines for future research.

While the extraction of the attributes needed to answer our research questions was conducted by the first author of this paper, the limitation field was filled out by all the authors, who individually reasoned and reviewed all the papers in order to find possible limitations. The individual considerations were then merged together by the first two authors.

3. Results

Before reporting the results of the SLR with respect to the considered research questions, in this section, we provide a brief overview of the demographics of the papers which passed the inclusion/exclusion criteria and the quality assessment.

3.1. Demographics

Table 4 reports the final list of relevant primary studies that were analyzed in this SLR, while column 'Year' and 'Publication Type' reporting the number of papers published in journals and conferences over the years. As it is possible to observe, all the considered papers were published between 2005 and 2017; 60% of these primary studies were published after 2012, possibly highlighting a growing trend that is now in the process of becoming a more established discipline. Moreover, we observed that 67% of the primary studies were published in conference proceedings, while the remaining 33% in international journals: this data seems to remark that articles published so far mostly report preliminary insights on how machine learning can be exploited for code smell detection.

Looking more in-depth to the authors of the papers in Table 4, we noticed that almost 60% of them was co-authored by researchers coming from two specific research groups, i.e., the ones at the University of Milano-Bicocca (Italy) and the École Polytechnique de Montréal (Canada), which indeed represent the current institutions of the researchers who contributed most to this research area, i.e., Arcelli Fontana[2] and Khohm[3], respectively. Thus, analyzing the primary studies we could derive two main expert institutions that can be considered the reference for young researchers interested in approaching the topic.

> **In Summary.** Looking at the number and types of papers published by the research community, we concluded that the adoption of machine learning techniques for code smell detection still possibly presents open challenges. Furthermore, we identified two specific research groups that can be considered as a reference for researchers interested in embracing the topic.

3.2. RQ1 - Code Smells Considered

In our SLR, we found that previous research focused on the 20 different code smells listed in Table 5, along with the frequency of appearance in the primary studies. As it is possible to observe, the *God Class* smell is the one considered most, i.e., 11 of the primary studies analyzed it. This result was

[2] `https://scholar.google.it/citations?user=8Z20aIMAAAAJ.`

[3] `https://scholar.google.com/citations?user=YYXb3KIAAAAJ.`

somehow expected, since this smell historically received more attention than others [48], other than being one of the most harmful for developers [31, 37].

Other smells frequently considered are *Long Method* with 6, *Feature Envy* with 5, and *Functional Decomposition* and *Spaghetti Code* with 4 references in the primary studies. The reason here is strictly connected with the research methodology adopted by the authors of these papers: as better explained later in Section 3.3.2, most of the previous studies relied on automated detectors to identify the dependent variable of the prediction models experimented. Therefore, the choice of the code smells to analyze was somehow driven by the availability of detectors. One of the most used tools is DECOR [40], which is indeed able to identify the code smells that were considered most in the primary studies.

Code smells like *Data Class*, *Swiss Army Knife*, and *Duplicated Code* were also considered a few times, while the appearance of the remaining ones can be considered occasional. Thus, we can conclude that the existing literature did not consider a variety of other code smells [6, 70], e.g., the *Inappropriate Intimacy* [6], that might also be potentially harmful to developers [31, 37] and for which more automated support would be needed. As a matter of fact, the use of machine learning for code smell detection only represents a partial solution that cannot be still extensively adopted by developers.

Interestingly, in the primary studies analyzed we found only one approach able to identify 12 code smells simultaneously, whereas the others limit their detection to at most 3 smell types. This means that developers can only rely on machine learners that cope with a limited set of code smells.

> **Summary for RQ1.** *God Class*, *Long Method*, *Functional Decomposition*, and *Spaghetti Code* are the smells more considered in the context of machine-learning based detection. Nevertheless, a large variety of other code smells from both the catalogs by Fowler [6] and Brown et al. [70] have not been considered. So, overall, we found that the research community only provided limited support for the identification of code smells through machine learning models.

3.3. RQ2 - Machine Learning Setup

The second research question of our SLR was related to the way machine learners used in literature were configured. Specifically, we aimed at understanding (i) independent and dependent variables considered, (ii) classifiers exploited, and (iii) training strategies adopted. The following subsections detail the results of our analyses.

Table 5: Code smells identified via SLR mentioned in the previous code smell detection literature.

S.No	Code Smell	Frequency	Cited
1	God Class	11	[S01], [S02], [S03], [S04], [S06], [S08], [S09], [S10], [S11], [S12], [S13]
2	Long Method	6	[S01], [S05], [S09], [S10], [S11], [S13]
3	Feature Envy	5	[S01], [S05], [S10], [S11], [S13]
4	Spaghetti Code	4	[S04], [S06], [S08], [S12]
5	Functional Decomposition	4	[S04], [S06], [S08], [S12]
6	Data Class	3	[S01], [S10], [S13]
7	Swiss Army Knife	3	[S06], [S08], [S09]
8	Duplicated Code	3	[S07], [S14], [S15],
9	Lazy Class	2	[S09], [S11]
10	Long Parameter List	2	[S05], [S09]
11	Message Chain	2	[S05], [S09]
12	Antisingleton	1	[S09]
13	Class Data should be Private	1	[S09]
14	Complex Class	1	[S09]
15	Refused Parent Bequest	1	[S09]
16	Speculative Generality	1	[S09]
17	Delegator	1	[S11]
18	Middle Man	1	[S05]
19	Switch Statement	1	[S05]
20	Large Class	1	[S13]

3.3.1. RQ2.1 - Independent Variables

The independent variables of a machine learning model, also called *features* or *predictors*, play a significant role to enable good prediction performance. To properly analyze the types of predictors used in the primary studies, we firstly grouped metrics according to their design goal. Specifically, we followed the metric classification framework provided by Pressman [71] and Nunez et al. [72] to categorize the independent variables exploited by the primary studies. For instance, the *Lines of Code* (LOC) [73] metric was assigned to the *Size* category since it aims at measuring the length of a source code file, whereas the *Lack of Cohesion Between Methods* (LCOM) [74] was considered as a *Cohesion* metric.

This process was conducted for each primary study. The number of studies which relied on each of the identified categories is shown in Figure 2. As a

Table 6: Independent variables grouped into categories and list of metrics in each category

S.No	Category	Metrics	Studies	Smells Detected
1	Size	LOC, LOCNAMM, NOM, NOPK, NOCS, NOMNAMM, NOA	[S01-2013], [S10-2017], [S13-2016]	God Class, Data Class, Long Method, and Feature Envy
	Complexity	CYCLO, WMC, WMCNAMM, AMWNAMM, AMW, MAXNESTING, WOC, CLNAMM, NOP, NOAV, ATLD, NOLV		
	Coupling	FANOUT, FANIN, ATFD, FDP, RPC, CBO, CFNAMM, CINT, CDISP, MaMCL, MeMCL, NMCS, CC, CM		
	Encapsulation	LAA, NOAM, NOPA		
	Inheritance	DIT, NOI, NOC, NMO, NIM, NOII		
	Cohesion	LCOM5, TCC		
	Others	NODA , NOPVA, NOPRA, NOFA, NOFSA, NOFNSA, NONFNSA, NOSA, NON-FSA, NOABM, NOCM, NONCM, NOFM, NOFNSM, NOFSM, NONFNABM, NONFNSM, NONFSM, NODM, NOPM, NOPRM, NOPLM, NONAM, NOSM		
2	Size	LOC, LOC_1, MLOCsum, NAD, NADExtended, NMA, NMD, NMDExtended, NOM	[S09-2015]	Antisingleton, God Class, Class Data Should Be Private, Complex Class, Large Class, Lazy Class, Long Method, Long Parameter List, Message Chains, Refused Parent Bequest, Speculative Generality, and Swiss Army Knife
	Complexity	WMC, McCabe, NOF, NOP, NOParam, Vgsum, WMC1, WMC_New		
	Coupling	CBO, RFC, CA, CE, CAM, ACAIC, ACMIC, DCAEC, DCC, DCMEC, IR, NCM, NOTI, RFC_New, connectivity		
	Encapsulation	DAM		
	Inheritance	DIT, NOC, MFA, AID, CLD, DIT_1, ICHClass, NMI, NMO, NOA, NOC_1, NOD, NOH, NOPM		
	Cohesion	LCOM, LCOM3, LCOM1, LCOM2, LCOM5, cohesionAttributes		
	Others	NPM, MOA, IC, CBM, AMC, DSC, NOTC, SIX		
3	Size	NAD, NADextended, NCP, NMA, NMD, NMDextended, NOM, PP	[S04-2010]	God Class, Functional Decomposition, and Spaghetti Code
	Complexity	CIS, McCabe, NOParam, WMC1, WMCinvocations, WMCmccabe		
	Coupling	ACAIC, ACMIC, CAM, CBO, CBOingoing, CBOoutgoing, connectivity, CP, DCAEC, DCCdesign, DCMEC, FanOut, NCM, RFP, RTP		
	Encapsulation	DAM		
	Inheritance	AID, ANA, CLD, DIT, EIC, EIP, ICHClass, MFA, NMI, NMO, NOA, NOC, NOD, NOH, NOP, NOPM, PIIR		
	Cohesion	cohesionAttributes, LCOM1, LCOM2, LCOM5		
	Others	DSC, MOA, REIP, RPII, RRFP, RRTP, SIX		
4	Size	LOC, NAD, NADextended, NMA, NMD, NMDextended, NOM	[S06-2012], [S08-2012]	God Class, Functional Decomposition, Swiss Army Knife and Spaghetti Code
	Complexity	McCabe, NOParam, WMC, WMC1, CIS		
	Coupling	ACAIC, ACMIC, CAM, CBO, CBOingoing, CBOoutgoing, connectivity, DCAEC, DCMEC, IR, NCM, NOTI, RFC, DSC, DCC		
	Encapsulation	DAM, NPrM		
	Inheritance	AID, ANA, CLD, DIT, ICHClass, MFA, NMI, NMO, NOA, NOC, NOD, NOH, NOP, NOPM		
	Cohesion	cohesionAttributes, LCOM1, LCOM2, LCOM5		
	Others	MOA, SIX, USELESS		
5	Size	NA, NC, NM, NO, ACT, COMP, NS	[S05-2011]	Lazy Class, Feature Envy, Middle Man, Message Chains, Long Method, Long Parameter Lists and Switch Statement
	Complexity	RFC, WAC, WMA, NP		
	Coupling	CBC		
	Encapsulation	AHF, AIF, CF, MHF, MIF, PF		
	Inheritance	DIT, NOC, NAI, NOI		
	Class Employment	C_PARAM		
	Diagrams	D_APPEAR		
	Relationship	ABSTR_R, ASSOC_R, DEPEND_R		
6	Size	No of instance variables of a class, Median of the number of statements of all methods of a class, Median of complexities of all methods of a class	[S11-2005]	Lazy Class, God Class, Delegator, Long Method, Feature Envy
	Complexity	NOP, NOLV		
	Coupling	No of internal connected components, No of external connected components		
7	Size	LOC	[S07-2012]	Duplicate Code
	Coupling	Number of Invocations, Number of Library Invocations, Number of Local Invocations, Number of Other Invocations, Number of Field Accesses		
	Process	History Features (Existence-Time, Number of Changes, Number of Recent Changes, File Existence-Time, Number of File Changes, Number of Recent File Changes), Destination Features (Whether it is a Local Clone, Fine Name Similarity, Masked File Name Similarity, Method Name Similarity, Sum of Parameter Similarities, Maximal Parameter Similarity, Difference on Only Postfix Number)		
8	Size	NMD, NAD	[S02-2009], [S03-2009], [S12-2011]	God Class, Functional Decomposition, and Spaghetti code
	Encapsulation	NAM (Number of Accessor Method)		
	Cohesion	LCOM5		
	Textual	ControllerClass rule (a controller class can be identified by its name or its method names, which must contain terms indicative of procedural programming: Process, Control, . . .)		
9	Textual	Tokenized source code e.g. terms of the class and programming constructs used in the class etc.	[S14-2016], [S15-2015]	Duplicate Code

result, we found that previous studies have almost never used a single group of software metrics except for two studies [S14] and [S15], but rather they

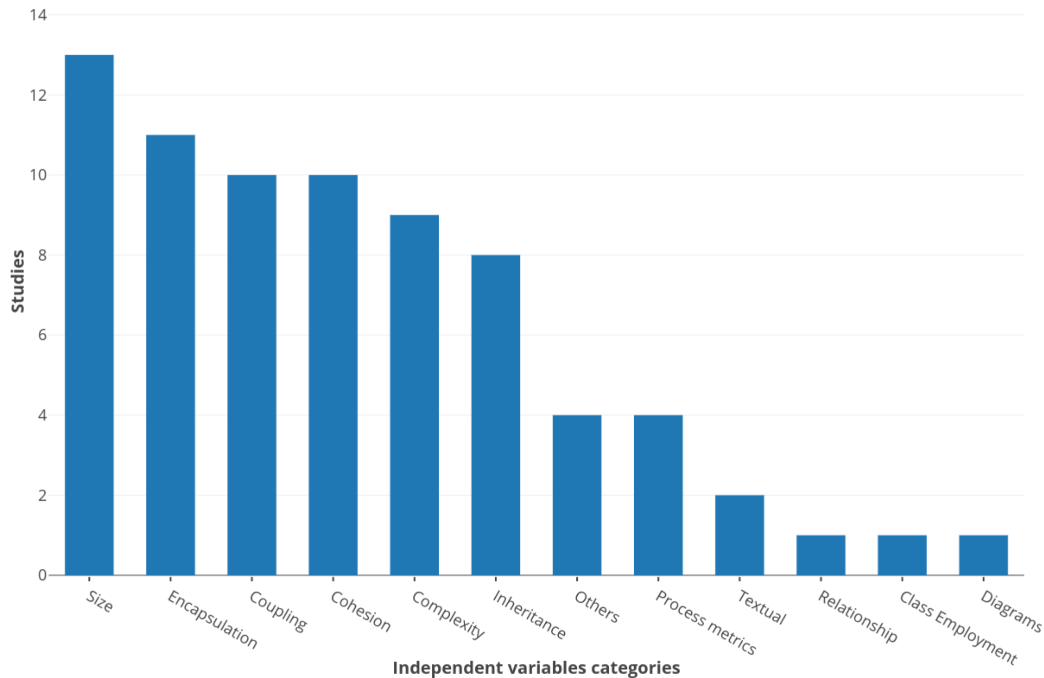

Figure 2: Bar chart reporting the number of primary studies adopting each of the metric category identified.

adopted, overall, nine different combinations of metrics. Table 6 reports the whole list of metric combinations: for each of them we also report (i) information about the specific metrics belonging to that combination, (ii) the primary studies adopting it, (iii) the total number of features exploited, and (iv) the code smell types detected by relying on it. Note that we also had a category called *Others*, that includes the metrics that are out from the considered metric classification framework: for instance, the feature *DSC*, i.e., Number of top-level entities in a model, was included in this category. All the metrics given in Table 6 are defined in AppendixB.

Looking deeper into the results, we basically observed how most of the code smell prediction models built so far relied on source code metrics able to capture the structural characteristics of a piece of code. The Chidamber and Kemerer (CK) metrics [74] were the most popular ones and were used by almost all the primary studies. The CK metrics suite originally consists of 6 metrics calculated for each class: WMC, DIT, NOC, CBO, RFC, and LCOM1. Likely, such a test suite was mostly selected because it con-

tains metrics that can capture different aspects of source code, e.g., cohesion, coupling or complexity. Thus, they represented the starting point to build machine learning approaches for code smell detection. The frequency of each metrics used across the studies is illustrated in Figure 3.

Besides them, a variety of other structural-based metrics were used to complement the assessment doable with the CK suite. For instance, the primary studies [S01], [S10], and [S13] customized some existing metrics to better capture the structure of source code elements: in particular, they created alternative versions of CK metrics that do not account for the accessor methods (*getters* and *setters*).

Other metrics such as the ones falling into the *Encapsulation* and *Inheritance* categories were instead considered as predictors starting from 2009 and 2011 respectively. This likely indicates the willingness of the research community to explore new metrics for improving the prediction performance of code smell detectors based on machine learning.

To broaden the scope of the discussion, our results clearly indicate the absence of studies investigating metrics different from the structural ones, that have been shown to be pretty effective when employed for code smell detection [44, 43]. Indeed, we found only one primary study relying on a combination of product and process metrics (i.e., [S07]): nevertheless, it aimed at predicting the existence of a single smell, i.e., *Duplicate Code*. Similarly, the use of textual-related information is rather scarce. Even though five primary studies relied on it ([S02], [S03], [S12], [S14], and [S15]), they adopted very basic textual metrics such as (i) string tokenization to identify *Duplicate Code* instances ([S14] and [S15]) or (ii) rules based on naming conventions to detect *God Class*, *Functional Decomposition*, and *Spaghetti Code* smells ([S02], [S03], and [S12]). Given the strong complementarity of such metrics [43, 33], we believe that their wider adoption could provide substantial improvements in the way machine learning approaches perform. At the same time, none of the surveyed studies adopted further alternative metrics like dynamic metrics [75] or metrics able to identify separation of concerns [76]. Intuitively, such metrics can perfectly fit the detection of many code smell types. For instance, dynamic metrics might be nicely adopted for the detection of *Message Chains* instances, whose definition is related to methods performing a long series of method calls. At the same time, concern separation metrics are already considered effective in the detection of code smells like *Divergent Change* and *God Class* [76], as they can quantify properties of concerns such as scattering and tangling [77]. We believe that the inclusion of those

Figure 3: Frequency of each metric, belongs to various categories, used across the studies

metrics can significantly boost the performance of existing approaches and, perhaps more importantly, they can be adopted to detect more code smells at different levels of granularity. As an example, we envision the adoption of concern separation metrics for the detection of code smells characterizing poor modularization, e.g., *Promiscuous Package* [78].

Perhaps more importantly, none of the previous studies took into consid-

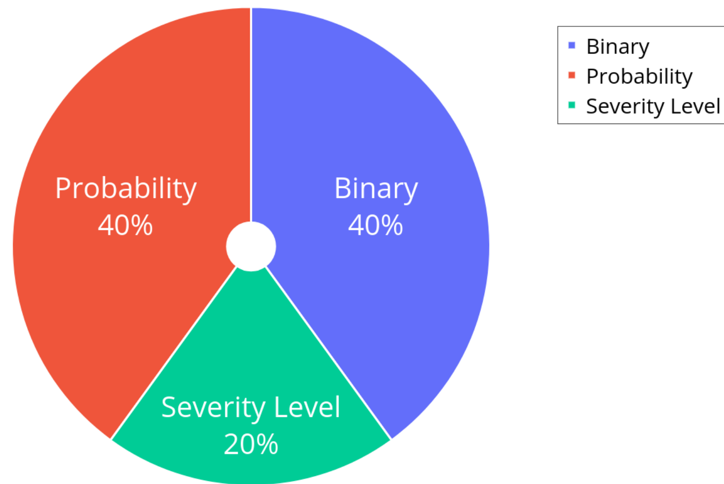

Figure 4: Pie chart showing the classification types used across the primary studies

eration developer-related information (e.g., developer's experience or workload) that have been successfully exploited in other software engineering contexts such as, e.g., bug- or change-prediction [79, 80, 81, 82]. When it turns to code smell detection, such factors might be even more important given the subjectivity with which they are interpreted by developers: in other words, they might be exploited with the aim of recommending to developers the code smells that are more critical for them based on their expertise or their workload in a certain moment. Thus, we believe that their integration represents an important opportunity for researchers in order to eventually bridge the gap between the research on the topic and the practice.

Summary for RQ2.1. Most of the primary studies analyzed relied on product metrics. The CK metric suite is the most used one. Nevertheless, we observed a noticeable lack of studies investigating the usefulness of process and textual metrics, which have been shown to be a good alternative source of information for detecting code smells. At the same time, we noticed the complete absence of the analysis of developer-related metrics, which might represent an important opportunity to reduce the subjectivity in the interpretation of code smells.

3.3.2. RQ2.2 - Dependent Variable

The dependent variable of a code smell prediction model provides information about the smelliness of a software artifact. A classification type defines the kind of output produced by a code smell prediction model. Figure 4 depicts the different classification types that have been used by the primary studies. As it is possible to observe, the *Binary* and *Probability* categories are the ones more used. In the former scenario, a model simply aims at discriminating whether a software artifact is affected by a smell or not, whereas in the latter one the goal is to predict the likelihood—usually in the range between 0 and 1—of a source code entity to be smelly. The basic difference between the two classification methods is related to the selection of a threshold, that in the *Binary* context is used to distinguish those elements affected by code smells.

Besides them, the *Severity Level* classification was used by 20% of the primary studies. In this context, the dependent variable is represented by a nominal value reporting the intensity of a design issue. For instance, in source [S10] the authors used a four-level scale: (i) absence of a smell, (ii) presence of a non-severe smell, (iii) presence of a smell, and (iv) presence of a critical smell. Table 7 shows the classification types which have been used by previous models to predict the 20 code smells identified via this SLR. The smelliness of all the code smells except Large Class have been predicted using Binary classification. Probability classification has been used to predict only six code smells. Only four code smells God Class, Long Method, Feature Envy and Data Class have been experimented with all the classification types.

Thus, we could recognize the presence of three different versions of the problem of code smell detection via machine learning techniques. However, given some recent results on the costs and risks of code smell removal [83], we believe that the *Severity Level* classification, based on independent variables that include a larger variety of metrics, has not reserved the right attention from the research community. Therefore, we recommend more empirical research on this topic.

Table 7: Code smells identified by models in the literature using various classification types.

(a)

Classification Type	God Class	Long Method	Functional Decomposition	Spaghetti Code	Feature Envy	Data Class	Swiss Army Knife	Duplicated Code	Lazy Class	Long Parameter List
Binary	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Probability	Y	Y	Y	Y	Y	Y	N	Y	N	N
Severity levels	Y	Y	N	N	Y	Y	N	N	N	N

(b)

Classification Type	Message Chain	Antisingleton	Class Data should be Private	Complex Class	Refused Parent Bequest	Speculative Generality	Delegator	Middle Man	Switch Statement	Large Class
Binary	Y	Y	Y	Y	Y	Y	Y	Y	Y	N
Probability	N	N	N	N	N	N	N	N	N	N
Severity levels	N	N	N	N	N	N	N	N	N	Y

Summary for RQ2.2. We identified three classification types: (i) the binary one, (ii) based on probability, and (iii) based on severity. The latter was involved in 20% of the primary studies, while the others were employed in 40% of the cases each. Based on our findings, we believe that more research on the severity-based classification is needed so that developers can better assess the harmfulness of code smells before their removal.

3.3.3. RQ2.3 - Machine Learning Algorithms

Our review showed that a large variety of machine learning algorithms, i.e., 17, were used for code smell detection. Figure 5 depicts the bar chart reporting the algorithms used along with the frequency of appearance in the primary studies. Note that in a single primary study, more classifiers might have been experimented.

As it is possible to see, Decision Trees were investigated by 6 primary studies. A possible reason lies into the output of this type of models, which consists of a rule indicating the conditions making a source code element smelly or not (i.e., a set of predicates aggregated by means of AND/OR operators): such an output is pretty *easy* to interpret, giving the opportunity to properly understand the mechanisms that lead to the detection of a certain code smell instance [84].

Support Vector Machines were also used a number of times. As previously reported [85], this classifier can achieve very high performance. At the same time, it is among the ones that are more complicated to configure and use in practice [86].

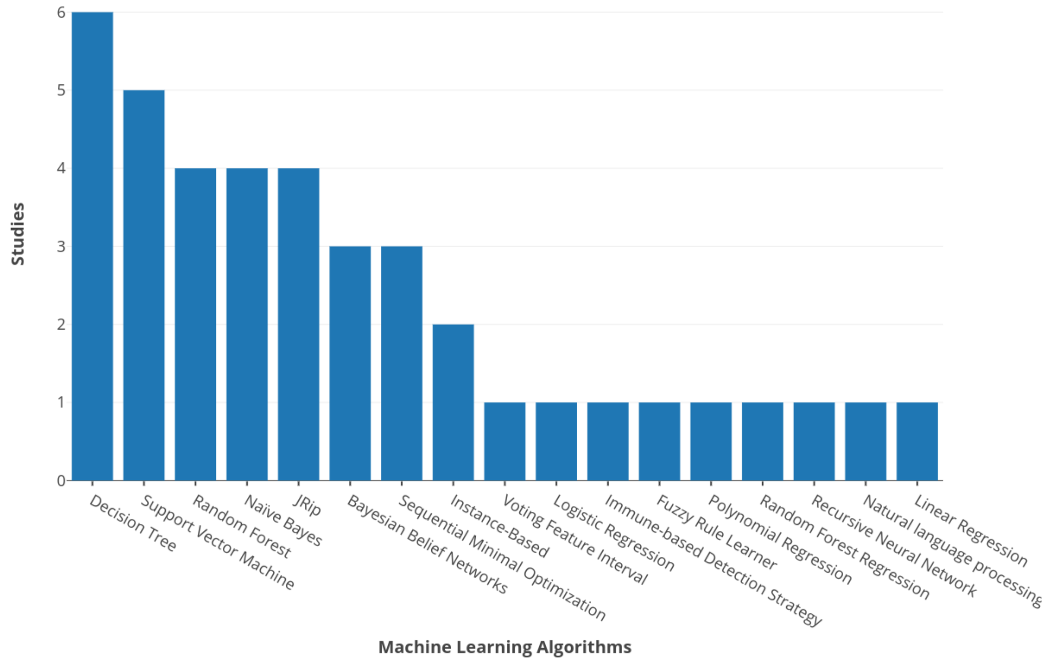

Figure 5: Frequency of each machine learning algorithm across the primary studies

As for Random Forest, it was used 4 times. Also, in this case, a likely reason behind its popularity is due to the good performance that this classifier ensures [87]. Note that we distinguished between decision trees and Random Forest since the latter can be considered as an ensemble method [88] rather than a standard classifier.

Other popular machine learning algorithms were Naive Bayes and JRip, while the remaining ones were used in a lower number of primary studies.

To sum up, the research community experimented a large variety of classifiers. However, we noticed a lack of studies that investigate the potential of ensemble techniques, which might notably improve the prediction capabilities of code smell models [88, 89]. Indeed, several other ensemble mechanisms were devised and tested in the last decade besides Random Forest [88]. Panichella et al. [58] study confirmed that CODEP (COmbined DEfect Predictor) have achieved higher prediction accuracy than the stand-alone defect predictors.

Another interesting observation can be done by considering the number of primary studies that explicitly considered the problem of configuring the

machine learning algorithm. As widely demonstrated in previous studies (e.g., in the study by Thomas et al. [90] and Tantithamthavorn et al. [91]), the configuration of these algorithms can significantly improve or worsen the overall model performance. Unfortunately, we found that only the study [S10] addressed this issue by employing the Grid-search algorithm [92], which explores the parameter space to find an optimal configuration. Thus, we can conclude that the real role of classifiers is still to evaluate and more research on this can possibly produce more effective code smell prediction models.

Summary for RQ2.3. Most of the existing studies employed decision trees or SUPPORT VECTOR MACHINES as machine learning algorithm. However, the problem of finding an optimal configuration was not properly addressed. At the same time, ensemble techniques were almost never considered as a way to improve the performance of code smell prediction models.

3.3.4. RQ2.4 - Training Strategies

Figure 6 reports the results for **RQ2.4**. Note that in one case (related to [S11]) we could not properly establish the type of training strategy adopted because this information was missing in the primary study.

The performance of most of the code smell prediction models devised so far was analyzed in both a within- and cross-project settings, meaning that the same model was trained once using data coming from the same project under analysis and once using data from external projects. We found this a positive result: indeed, as recommended by He et al. [93] and Watanabe et al. [94] whenever a certain technique is experimented in a cross-project setting, it should be applied in a within- project setting as well in order to fairly benchmarking its real capabilities.

Only the study by Arcelli Fontana et al. ([S10]) explicitly targeted the cross-project scenario, while the remaining ones focused on the within-project one.

Given the practical complexity of building a dataset of annotated code smell instances, needed to train a within-project model, we believe that focusing more on cross-project code smell prediction represents the right choice to take.

Summary for RQ2.4. Most of the code smell prediction models were experimented in both a within- and cross-project setting, while 33.33% of them were only considered in a cross-project scenario.

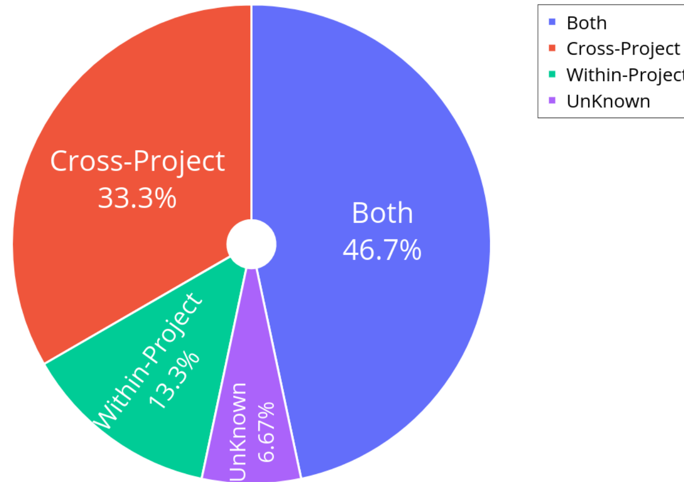

Figure 6: Pie chart showing the training strategies used across the primary studies

3.4. RQ3 - Evaluation Setup

Our third research question revolves around the code smell prediction model evaluation strategies. It basically refers to (i) validation techniques, (ii) assessment metrics adopted, and (iii) datasets exploited.

3.4.1. RQ3.1 - Validation Techniques

The validation methodologies adopted by the primary studies analyzed are shown in Figure 7.

The vast majority of them adopted the k-fold cross-validation [95]. Generally, k is set to 10. In this way, the strategy randomly partitions the original set of data into 10 equal sized subsets. Of the 10 subsets, one is retained as a test set, while the remaining 9 are used as a training set. The cross-validation is then repeated 10 times, allowing each of the 10 subsets to be the test set exactly once [95].

Although reasonable for an initial validation, the adoption of this strategy has two limitations that are likely to influence the results of all the primary studies using it. In the first place, the randomness with which the subsets are created strongly impact the performance of a prediction model [96]: to cope with it, the minimum requirement would have been that of repeating the validation n times, e.g., 100 times as recommended by Hall et al. [96]. Unfortunately, none of the primary studies considered this issue and, therefore,

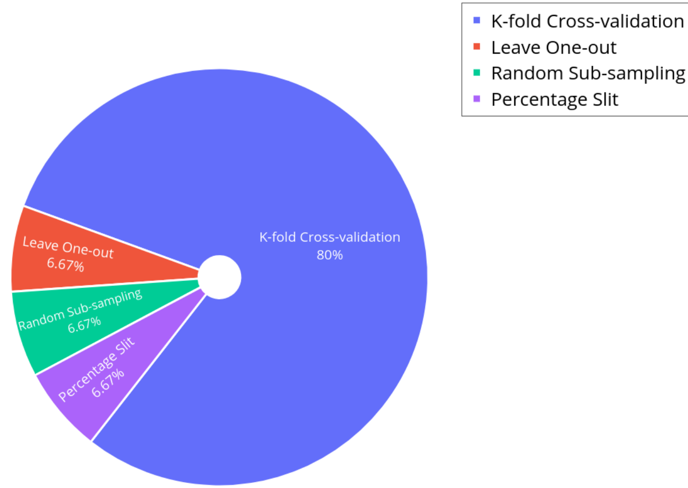

Figure 7: Pie chart showing the validation techniques used across the primary studies

their evaluations are strongly threatened.

Perhaps more importantly, a recent study by Tantithamthavorn et al. [60] demonstrated that this validation strategy is among the most biased and less reliable. Thus, its use is highly discouraged. To some extent, this means that all the primary studies relying on k-fold cross-validation should be re-evaluated in order to empirically analyze the impact of this issue on the achieved results.

Besides [S11] that adopted the more reliable Leave-one-out system validation [60], the remaining studies still adopted biased and error-prone methods. Thus, we conclude that the reliability of the findings provided so far on the ability of machine learning models to detect code smells is biased and should be re-evaluated.

Summary for RQ3.1. 93% of the primary studies analyzed relied on a biased validation strategy that likely led to interpretation errors. Thus, their empirical re-evaluation is needed.

3.4.2. RQ3.2 - Evaluation Metrics

Figure 8 shows the bar chart reporting the evaluation metrics adopted by the primary studies. The most widely used is *precision*, i.e., the ratio between true and false positives, while the *recall* was adopted by 10 of them.

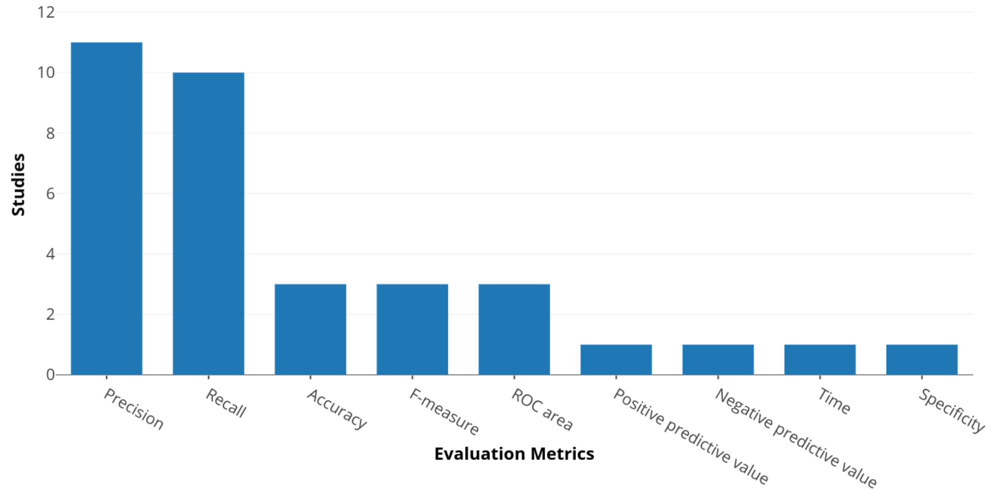

Figure 8: Evaluation metrics used across the primary studies

Other metrics were used less often, but in general, we can observe that the main target of the conducted evaluations was the overall accuracy of the prediction models. Only one study ([S14]) adopted *Time* to evaluate the training performance of the built model.

More in general, it is important to note that most of the previous studies relied on threshold-dependent metrics, i.e., metrics whose computation depend on the specific threshold assigned to the machine learner to discriminate smelly and non-smelly source code elements. As recently shown [96], these metrics can bias the interpretation of the performance of a prediction model and should be at least complemented by threshold-independent metrics like the Area Under the ROC Curve and the Matthew's Correlation Coefficient [96]. In this regard, we can claim that existing literature possibly missed an important piece of information while assessing the real capabilities of code smell prediction models.

Summary for RQ3.2. Most of the primary studies considered threshold-dependent metrics that possibly bias the interpretation of the achieved results. This result confirms that a proper assessment of existing code smell prediction models would be worthwhile.

3.4.3. RQ3.3 - Code Smells Datasets

The final step, to understand the evaluation setup of the primary studies, consisted of the analysis of the datasets exploited. As shown in a recent work [97], the dataset might influence the performance of machine learning models. Table 8 reports the list of systems considered in the years. The studies [S01], [S10], and [S13] conducted experimentations on the Qualitas Corpus dataset [98], which contains 112 open-source Java systems along with the code metric values measuring a large variety of structural aspects of the source code.

Besides them, all the other studies performed small-scale empirical assessments having an object up to 8 software projects. More importantly, only 3 studies ([S05], [S07], and [S11]) tested their approaches on industrial datasets.

Besides the size of the dataset, it is also important to analyze the way the dependent variable (i.e., the smelliness of code artifacts) of such datasets was identified. We observed that only the work by White et al. [S14] relied on a manually validated set of code clones; instead, all the other primary studies estimate the smelliness of source code artifacts by means of existing code smell detectors. For instance, Arcelli Fontana et al. [S13] firstly ran a set of detectors over the Qualitas Corpus dataset and then manually evaluate a sample of the code smell instances identified by such detectors: while this process allowed to not consider as smelly the false positive instances, the dataset could not deal with false negatives, i.e., actual smelly instances wrongly detected as non-smelly. Thus, such datasets could have missed some real instances. In other words, there is still a lack of evaluations conducted on totally manually built datasets containing a comprehensive set of code smells. Noticeably, some of them are already publicly available [99, 11].

Summary for RQ3.3. We obtained two main findings. On the one hand, a few studies were performed on a large scale, thus threatening the generalizability of the reported findings. On the other hand, we found only one primary study that investigated the performance of machine learning techniques on manually built datasets reporting a comprehensive set of code smells.

3.5. RQ4 - Performance Meta-Analysis

The last research question of our study was related to a statistical meta-analysis of the performance of code smell prediction models. Ideally, this

Table 8: Datasets used to train code smell detection models in the previous literature

S.No	Systems used in the dataset	Studies	Dataset type
1	Qualitas Corpus: 20120401r	[S01], [S10], [S13]	Open-source
2	GanttProject v1.10.2 and Xerces v2.7.0	[S03], [S04], [S12]	Open-source
3	ArgoUML v0.19.8, Azureus v2.3.0.6, and Xerces v2.7.0	[S06], [S08]	Open-source
4	Eclipse JDT and Xerces	[S02]	Open-source
5	Eclipse, Mylyn, ArgoUML, and Rhino	[S09]	Open-source
6	ANTLR 4, Apache Ant 1.9.6, ArgoUML 0.34, CAROL 2.0.5, dnsjava 2.0.0, Hibernate 2, JDK 1.4.2, JHotDraw 6	[S14]	Open-source
7	git v1.7.9-rc1, xz 5.0.3, bash 4.2, and e2fsprogs 1.41.14	[S15]	Open-source
8	Two large industrial software projects from Microsoft	[S07]	Industrial
9	Two small industrial software projects	[S11]	Industrial
10	Seven data sets from software of previous works of literature that have bad smells	[S05]	Industrial

kind of investigation would have required the complete re-execution of the prediction models on a common dataset and using common evaluation metrics, so that they might have benchmarked. However, this is outside the scope of a Systematic Literature Review: rather, the goal is to *synthesize* the results reported in the primary studies, accepting the limitations that we observed during the analysis of the previous research questions. In this study, we analyzed the impact of (i) independent variables, (ii) machine learning algorithm, and (iii) training strategy on the performance of code smell prediction models. We kept out of the scope of this analysis the validation technique because the vast majority of the previous studies applied the 10-fold cross validation (see Section 3.3.3); as a consequence, only a limited amount of data points is available for the other validation strategies, making not reliable an analysis like the one proposed herein.

A statistical meta-analysis aims at combining multiple studies in an effort to increase power over individual studies while improving the estimates of the effect sizes and resolve uncertainties when different studies disagree [100, 101]. Indeed, while individual studies are often too small for drawing reliable generalizable conclusions, their combination might provide less random error and narrower confidence intervals [102, 103]. Meta-analyses have also an important drawback: they cannot correct for the poor design and bias in the original studies [101]. Nonetheless, the advantages are far more valuable than the few downsides if the analyses are carried out and interpreted carefully. In the context of software engineering, Kitchenham et al. [104] endorsed the use of meta-analyses. To perform the meta-analysis, we followed the guidelines available in the book by Cooper et al. [100]; this is the same methodology already successfully adopted in previous meta-analyses conducted in the context of software engineering research [105, 106, 107].

The first step in a meta-analysis is that to compute, from each of the indi-

vidual studies, the outcome of interest and summary statistics. As explained in Section 3.4.2, not all the previous studies assessed the performance of the proposed models in the same way. To have a common basis and conduct a fair comparison, we only considered the papers that evaluated the code smell prediction models in terms of overall precision, recall, or F-Measure: we selected these metrics since they were used in most of the other sources. We used Comprehensive Meta-Analysis v2 [108] to calculate effect size estimates for all the F-Measure values in the primary studies. It is worth noting that, to be comparable across studies, effect sizes must be standardized. To this aim, we used Hedges' g as the standardized measure of effect size. Basically, this represents the difference between the outcome means of the treatment groups, but standardized with respect to the pooled standard deviation and corrected for small sample bias [108]. An effect size value of 0.5 indicates that the mean of the treatment group is half a standard deviation larger than the mean of the control group, while (i) effect sizes larger than 1 can be considered large, (ii) effect sizes between 0.38 and 1.00 medium, and effect sizes between 0 and 0.37 small [108].

In the second stage, individual studies are summarized. While different methods have been proposed in meta-analysis research [100], the *inverse-variance* method [109] is the most recommended and widely used [110]. It basically assigns a higher importance to larger studies and less to smaller ones. Moreover, as recommended by Cooper et al. [100], we defined fixed and random effect models. The former assumes the existence of one source of variability: the within-study error, for which the contribution of each study is proportional to the amount of information observed in that study. As a consequence, this implies that the differences among studies are solely due to sampling error, i.e., the size of a study. However, the assumption of only one source of variability might not always hold and, therefore, the second source of control is usually taken into account: this is the random effect (also known as *between-study error*), that assumes that the differences among individual studies are due to both sampling error and other variables/factors that have not been accounted for. Modeling the problem in this way, we could provide an overview of what are the factors influencing more the performance of the models proposed so far. To implement these analyses (including *inverse-variance* weighting schema and fixed/random effects), we relied on

the `forestplot` package[4] available in the R toolkit. In the following section, we report and discuss the results obtained when analyzing the forest plots.

3.5.1. RQ4.1 - Impact of Independent Variables

Figure 9 reports the forest plot of the performance of various code smell prediction models trained on different independent variables. The complete list of independent variables in each set (A-G) is given in AppendixC. The primary studies given in Figure 9 have either explicitly mentioned the precision, recall, and F-measure values or could be computed by us. F-measure value has been used to analyze the impact of the independent variables on the performance of code smell prediction models. When multiple code smell types have been considered by a certain primary study, we computed and reported in Figure 9 the average F-Measure obtained over the different smells. It is important to note that in our case the average can be considered as a valid aggregator because all the code smell types have been detected using the same set of independent variables and thus the average can actually represent the ability of certain variables in detecting different smells.

As it is possible to see, the overall performance, with 95% of the confidence interval, is ≈81%, meaning that the reported F-Measure of the primary studies is extremely high and overcome the general accuracy obtainable using heuristic-based approaches [48]. In this case, both fixed- and random-effects converge over a similar conclusion, without contrasting result: this means that the results hold independently from eventual confounding effects present in the individual studies. More in detail, our data reports that having a large number of metrics able to capture different aspects of source code significantly helps a code smell prediction model to perform better. Specifically, the models presented in the studies [S01] and [S13] were the ones using the largest amount of independent variables and that also included ad-hoc metrics such as the LOC without Accessor Methods (LOCNAMM). They reached the highest F-measure values, quantifiable in 91.29% and 97.44%, respectively. Similarly, the metrics set used in the study [S04] also consists of a number of different independent variables, leading the model to perform with an F-measure of 90.59%. The difference in the performance might reflect the importance of a good selection of metrics able to characterize the symptoms behind the presence of code smells. The independent variables in

[4] https://cran.r-project.org/web/packages/forestplot/forestplot.pdf

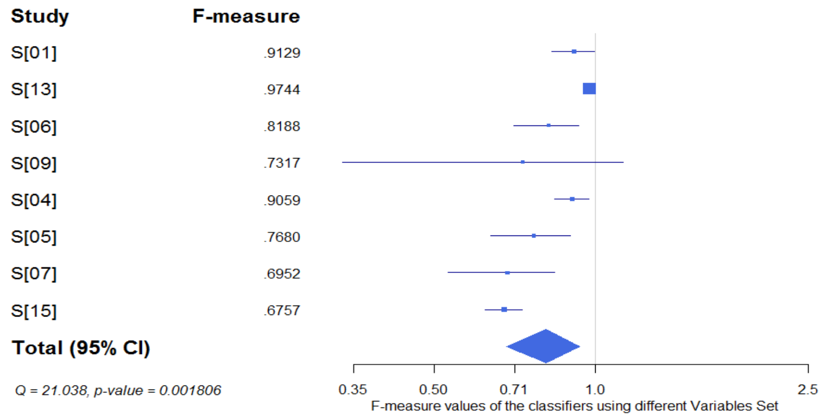

Figure 9: Performance of code smell prediction models based on the different combination of independent variables adopted by the primary studies.

the sets B, C, and D (studies [S06], [S09] and [S04]) are almost the same with small differences related to the number of chosen metrics; however, the difference in the performance of the models trained on these independent variables shows that the selection of independent variables has an effect on the performance achieved by the models. At the same time, the set E (study [S05]) has independent variables categories Diagrams, Relationship, and Class Employment that are unique and produced good performance. Finally, the independent variables in sets F and G were used to train models able to predict the Duplicate code smell by studies [S07] and [S15], respectively. The independent variables in set F perform better than the set G, that consists of source code text metrics whereas set F contains size, coupling and process metrics. This may suggest that textual-related information can provide additional and orthogonal information to improve the performance of existing models.

The observations made above allow us to address our research question: the selection of proper independent variables has an important impact on the performance of code smell prediction models. Based on our data, such an impact can be up to 29%.

Summary for RQ4.1. The proper selection of independent variables impacts the performance of code smell prediction models up to 29%. The specification of ad-hoc metrics able to capture the symptoms of specific code smell types seems to be a key factor for the creation of good prediction models.

3.5.2. RQ4.2 - Impact of Machine Learning Algorithm on Performance

Figure 10 depicts the forest plot showing the performance of the different machine learners adopted by the primary studies in the detection of code smells. We only analyzed five machine learners because for the rest of the machine learners the frequency is quite low and therefore they cannot be considered for robust analysis. As done for the previous analysis, in case multiple code smell types have been considered by a primary study, we computed and reported the average F-Measure obtained over the different machine learners considered. Also, in this case, both fixed- and random-effects did not provide contrasting results: with a confidence interval of 95%, the overall performance is ≈79%. From the plot, we observed that models based on JRip and Random Forest seem to provide better performance than the others. Similarly, models based on tree-based classifiers also performed well and could achieve an F-measure value up to 83%. The performance of SVM (Support Vector Machine) and Naive Bayes classifiers were found to be pretty limited when compared to the other classifiers. In this regard, it is worth mentioning that the SVM model was experimented by Fontana et al. in studies [S01] and [S13] and produced quite good performances in predicting code smells: likely, this difference was due to the steps conducted by the authors, which included parameter optimization and standardization; this may further suggest that such operations are vital to achieve better performance. At the same time, the dataset exploited in those studies was also the largest available one, with one-third smelly and two-third non-smelly instances, manually validated.

All in all, we can conclude that JRip and Random Forest are the most reliable classifiers for predicting code smells. A likely reason is concerned with their ability to automatically selecting the most important features to use in the prediction. However, our findings reveal that an appropriate selection of classifiers is important since this choice impacts by up to 40% the performance of code smell prediction models.

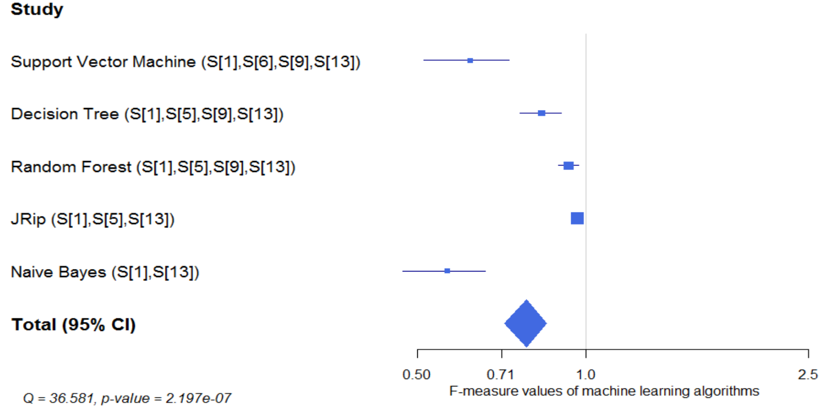

Figure 10: Performance of machine learning algorithms

Summary for RQ4.2. Among all the classifiers experimented by the primary studies, JRip and Random Forest were reported as the most effective. However, we discovered that a wrong selection of the machine learning algorithm to use impacts the performance of code smell prediction models by up to 40% in terms of F-Measure.

3.5.3. RQ4.3 - Impact of Training Strategies

In the case of training strategies, we could directly compare within- and cross-project models by extracting data from the primary studies that tested both of them. For this reason, we excluded from this analysis those studies not reporting experimentations in both the contexts. Our findings are reported in Figure 11. We could also report the Odds Ratio [111] (OR), a measure that quantifies how much the within-project models are better than the cross-project ones in terms of F-Measure: more specifically, this metric can estimate the extent to which one training strategy has been found in literature to be better than the other one. For instance, if the OR of the within-project strategy in comparison with the cross-project one is 1.10, this indicates that it is 10% better than the cross-project strategy.

As expected, within-project models constructed using data coming from the same project where the predictions are applied perform generally better than cross-project models. Overall, we observed that within-project models have performance about two times better than cross-project ones (OR=1.9). Likely, this is due to the well-known phenomenon of *data heterogeneity* [112],

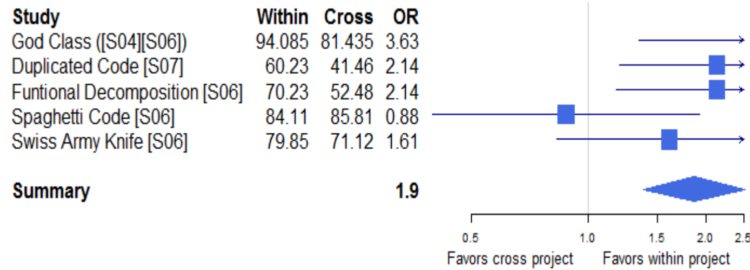

Figure 11: The impact of training strategy on the performance of CSD models.

i.e., data coming from external projects have a different metric distribution and cannot be easily used to train models that work on different datasets. To partially solve this problem, some approaches were proposed in the field of bug prediction (e.g., external data filtering approaches [113]): the assessment of such mechanisms in the context of code smell prediction models would be worthwhile. In the worst case, cross-project models have performance ≈19% lower than the one reported for within-project ones. The result holds for all the code smells considered by existing approaches. The only exception to this discussion is related to the *Spaghetti Code* smell: in this case, a cross-project strategy slightly performs better. We cannot speculate on the reasons behind this result since it might be due to several reasons. In our future research agenda, we plan to further study this point.

Summary for RQ4.3. Data heterogeneity plays a role in code smell prediction. We found that cross-project models are up to 19% less effective than within-project ones. We cannot conclude that within-project trained models are better. The higher performance may be attributed to the over-fitting of the model. It also explores that, the data drift problem has not been solved in machine learning based code smell detection.

4. Discussion and Implications

At the end of our analyses, in this section, we further discuss the main findings of our work as well as delineate guidelines and future trends that

Table 9: Code smell types shown as harmful for developers by previous research.

Name	Description
Class Data Should Be Private	A class exposing its attributes.
Inappropriate Intimacy	Two classes exhibiting high coupling between them.
Middle Man	A class delegating all its work to other classes.
Orphan Variable of Constant Class	A class containing variables used in other classes.
Refused Bequest	A class inheriting functionalities that it never uses.
Speculative Generality	An abstract class that is not actually needed, as it is not specialized by any other class.
Tradition Breaker	A class that does not fully extend the parent class and has no subclasses.

the research community might be interested in. For the sake of clarity, in the following we also reported the specific research question relating to the discussed point.

- **RQ1** - *Limited support to identify and prioritize code smells using machine learning techniques.* Looking at the types of code smells that have been subject of an investigation by researchers in the past ($\mathbf{RQ}_1$), we can clearly delineate a lack of machine learning-based automated solutions for the detection of code smells. Indeed, we showed that only a few design problems, *God Class*, *Long Method*, *Functional Decomposition*, and *Spaghetti Code*, have received some attention, while the capabilities of machine learning in detection of the vast majority of the code smells in the catalog by Fowler [6] and Brown et al. [70] are still not assessed or only preliminarily evaluated. More importantly, some of the non-supported smells, e.g., *Refused Bequest* [6], have been shown to be harmful to developers by recent studies [31, 114]: as a consequence, the first recommendation is: **start from the empirical investigations on code smell harmfulness when deciding which of them a machine learning approach should target**. To ease the work of researchers of the field, we summarized the studies on code smell perception in order to come up with a detailed list of design problems that need to be investigated further. This list is reported in Table 9: it is composed of seven code smells that are not related to source code complexity or misplacement of code elements. Rather, we observed that major attention should be given to **inheritance-related** and **coupling-related** code smells.

- **RQ2** - *Research in code smell prediction is stuck in a local optimum.* Most of the approaches defined so far rely on structural code metrics as independent variables. We believe that this represents an important

limitation that might possibly lead existing techniques to provide a sub-optimal support to practitioners. More specifically, recent research has shown how code metrics are not enough to characterize neither code smells [115, 116, 117] nor the developers' perception of design problems [33]. To overcome these limitations, the adoption of a broader set of metrics exploiting different types of information, i.e., textual, historical, dynamic, and concern separation aspects, could be beneficial to devise more effective solutions that are also closer to the way developers perceive and identify code smells. Thus, a clear opportunity for future research is to **investigate the impact of other sources of information for the classification of code smell instances**. Even more importantly, we believe there is a need for **developer-oriented** methods that take into account the developers' characteristics to perform predictions on code smells that are relevant to them.

- **RQ2** - *Prioritization approaches are still missing.* Only a small amount of works (20%) focused on the problem of code smell prioritization: as the intensity of code smells represents one of the key aspects when scheduling refactoring operations [31] and given the great potential that machine learning has in mitigating the subjectiveness with which code smells are interpreted [51], we highlighted a lack of studies that investigate how machine learning-based approaches can be adapted for prioritization purposes. Thus, we argue that **more specific tools that capture the actual severity of code smells would be needed** [118].

- **RQ2** - *Existing machine learning approaches were not properly configured.* In the context of our systematic literature review, we pointed out several aspects related to the settings of existing machine learning approaches that might have potentially biased the results achieved so far. A clear example is related to the configuration of the algorithms exploited (e.g., Support Vector Machine). We argue that a **closer look into the way machine learning techniques are configured** is needed to properly interpret their results.

- **RQ2** - *A more comprehensive overview of machine learning performance is needed.* We observed that a number of classifiers have been

used by previous research to predict the smelliness of source code. However, there is still a lack of understanding on **the role of ensemble techniques for code smell prediction**. As ensemble techniques applied to defect predictors have produced higher prediction accuracy than stand-alone defect predictors [58]. This is a clear opportunity for the research community in order to improve the detection of design issues [119].

- **RQ3** - *On the Validation of Machine Learning Techniques.* Our findings revealed that previous studies are affected by several threats to the validity with respect to the validation strategies used to measure the performance of the proposed models. In essence, we recommend the adoption of the guidelines by Hall et al. [96] for the correct setup of empirical investigations. Particular attention should be devoted to the **interpretation of the performance of machine learning models** through threshold-independent metrics and the **application of preprocessing techniques** such as feature selection and data balancing to correctly set up the machine learner. Finally, we observed the need for **manually validated dataset** reporting the actual set of code smells affecting software systems.

- **RQ4** - *On the Impact of Machine Learning Settings.* The meta-analysis we performed highlighted that each of the treated aspects, i.e., independent variables, machine learning algorithm, and training strategy, might have a considerable impact on the performance of machine learning approaches. This is a key aspect to take into account for future research: a **proper assessment of the impact of each aspect related to the settings of machine learning approaches should be performed**, in order to ease the interpretation of the performance as well as the sensibility of the technique to different settings.

We believe that each of the points above deserves dedicated studies and research. At the same time, we call for more sound and rigorous data analysis processes.

5. Threats to Validity

The main threats affecting the validity of our SLR are related to the way we selected and extracted the data used for our analyses.

The most important challenge for any SLR is related to the identification of an effective and complete set of search terms. To mitigate threats in this regard, we defined a detailed search strategy: once we had extracted the first set of keywords from the research questions we posed, we identified synonyms or alternative spellings and then we verified the presence of such terms in the relevant papers. Moreover, it is important to note that the steps leading to the selection of the relevant papers were double-checked by the second author of this paper. In addition to the automated search in the publication databases selected, we also performed snowballing in order to find other relevant papers. Finally, we conducted an additional manual search over the papers published at the top software engineering venues in the last ten years.

The study selection process was carried out by applying the exclusion and inclusion criteria. It was performed in two steps: firstly, the first author of the paper applied the criteria and filtered out non-relevant papers and then the second author double-checked the selection process. As a result, the agreement between them was very high: this somehow confirms the validity of the whole process.

As for the quality assessment and data extraction process, we set up a formal procedure leading to the definition of (i) a checklist for verifying the presence of the needed information in the selected publications and (ii) a data extraction form to gather information that allowed us to answer our research questions.

Other threats are related to the meta-analysis performed when comparing the performance reported in the primary studies. In doing such an analysis, we employed fixed and random effect models [103] based on the F-measure of the approaches proposed in the literature. While this can be considered as a standard methodology to combine and synthesize the results of individual studies [103], it is important to note that, by nature, this method cannot correct for the poor design and bias in the original studies [101]. We accepted this limitation with the aim of providing a general overview of the impact of independent variables, machine learning algorithms, and training strategy on the performance of code smell prediction models. A detailed benchmark analysis featuring the re-execution of all prediction models on a common dataset is part of our future research agenda and would definitively strengthen the conclusions provided by this study.

6. Conclusion

This paper reported a Systematic Literature Review on the use of machine learning techniques for code smell detection. It targeted four specific aspects related to how previous research conducted experimentations on code smell prediction models, i.e., (i) which code smells have been considered, (ii) what has been the machine learning setup adopted, (iii) which types of evaluations strategies have been exploited, and (iv) what are the claimed performance of the proposed machine learning models. Our work was conducted on papers published between 2000 and 2017. From an initial population of 2,456 papers, we analyzed 15 of them proposing machine learning approaches for code smell detection.

Our analyses highlighted a number of limitations of existing studies as well as open issues that need to be addressed by future research. We also provided a list of actions to perform in order to make the research field more concrete. In this respect, we hope that our work will provide a reference point for conducting future research and the recommendations provided will lead to higher quality research in this area.

7. Acknowledgment

This work is supported by National Natural Science Foundation of China under Grant Nos. 61432001. The first author of this paper is sponsored by the CAS-TWAS President's Fellowship for International Ph.D. students. Palomba gratefully acknowledges the support of the Swiss National Science Foundation through the SNF Project No. PP00P2_170529.

8. References

[1] M. M. Lehman, Programs, life cycles, and laws of software evolution, Proceedings of the IEEE 68 (9) (1980) 1060–1076.

[2] D. A. Tamburri, F. Palomba, A. Serebrenik, A. Zaidman, Discovering community patterns in open-source: a systematic approach and its evaluation, Empirical Software Engineering (2018) 1–49.

[3] D. L. Parnas, Software aging, in: Proceedings of the 16th international conference on Software engineering, IEEE Computer Society Press, 1994, pp. 279–287.

[4] P. Avgeriou, P. Kruchten, I. Ozkaya, C. Seaman, Managing technical debt in software engineering (dagstuhl seminar 16162), in: Dagstuhl Reports, Vol. 6, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

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

[6] M. Fowler, B. Kent, B. John, O. William, R. Don, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

[7] P. Kruchten, R. L. Nord, I. Ozkaya, Technical debt: From metaphor to theory and practice, Ieee software 29 (6) (2012) 18–21.

[8] C. Seaman, Y. Guo, Measuring and monitoring technical debt, Advances in Computers 82 (25-46) (2011) 44.

[9] M. Abbes, F. Khomh, Y.-G. Gueheneuc, G. Antoniol, An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension, in: Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering, CSMR '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 181–190. doi:10.1109/CSMR.2011.24.

[10] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, G. Antoniol, An exploratory study of the impact of antipatterns on class change- and fault-proneness, Empirical Software Engineering 17 (3) (2012) 243–275. doi:10.1007/s10664-011-9171-y.

[11] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, A. De Lucia, On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation, Empirical Software Engineering (2017) 1–34.

[12] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, A. Bacchelli, On the relation of test smells to software code quality, in: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2018, pp. 1–12.

[13] L. Pascarella, F. Palomba, A. Bacchelli, Fine-grained just-in-time defect prediction, Journal of Systems and Software.

[14] R. D. Banker, S. M. Datar, C. F. Kemerer, D. Zweig, Software complexity and maintenance costs, Commun. ACM 36 (11) (1993) 81–94. doi:10.1145/163359.163375.

[15] A. Yamashita, L. Moonen, Do code smells reflect important maintainability aspects?, in: 2012 28th IEEE International Conference on Software Maintenance (ICSM), 2012, pp. 306–315. doi:10.1109/ICSM.2012.6405287.

[16] D. I. K. Sjøberg, A. Yamashita, B. C. D. Anda, A. Mockus, T. Dybuantifying the effect of code smells on maintenance effort, IEEE Transactions on Software Engineering 39 (8) (2013) 1144–1156. doi:10.1109/TSE.2012.89.

[17] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, D. Poshyvanyk, When and why your code starts to smell bad (and whether the smells go away), IEEE Transactions on Software Engineering.

[18] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, D. Poshyvanyk, An empirical investigation into the nature of test smells, in: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ACM, 2016, pp. 4–15.

[19] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, A. De Lucia, A large-scale empirical study on the lifecycle of code smell co-occurrences, Information and Software Technology 99 (2018) 1–10.

[20] R. Arcoverde, A. Garcia, E. Figueiredo, Understanding the longevity of code smells: preliminary results of an explanatory survey, in: Proceedings of the 4th Workshop on Refactoring Tools, ACM, 2011, pp. 33–36.

[21] A. Chatzigeorgiou, A. Manakos, Investigating the evolution of bad smells in object-oriented code, in: Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the, IEEE, 2010, pp. 106–115.

[22] A. Lozano, M. Wermelinger, B. Nuseibeh, Assessing the impact of bad smells using historical information, in: Ninth international work-

shop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting, ACM, 2007, pp. 31–34.

[23] R. Peters, A. Zaidman, Evaluating the lifespan of code smells using software repository mining, in: Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on, IEEE, 2012, pp. 411–416.

[24] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, A. De Lucia, On the impact of code smells on the energy consumption of mobile applications, Information and Software Technology 105 (2019) 43–55.

[25] C. Vassallo, F. Palomba, A. Bacchelli, H. C. Gall, Continuous code quality: are we (really) doing that?, in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ACM, 2018, pp. 790–795.

[26] C. Vassallo, F. Palomba, H. C. Gall, Continuous refactoring in ci: A preliminary study on the perceived advantages and barriers, in: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2018, pp. 564–568.

[27] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, A. Zaidman, H. C. Gall, Context is king: The developer perspective on the usage of static analysis tools, in: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2018, pp. 38–49.

[28] L. Pascarella, F. Palomba, M. Di Penta, A. Bacchelli, How is video game development different from software development in open source?

[29] L. Pascarella, F.-X. Geiger, F. Palomba, D. Di Nucci, I. Malavolta, A. Bacchelli, Self-reported activities of android developers, in: 5th IEEE/ACM International Conference on Mobile Software Engineering and Systems, New York, NY, 2018.

[30] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, F. Palomba, An experimental investigation on the innate relationship between quality and refactoring, Journal of Systems and Software 107 (2015) 1–14.

[31] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, Do they really smell bad? a study on developers' perception of bad code smells, in: Software maintenance and evolution (ICSME), 2014 IEEE international conference on, IEEE, 2014, pp. 101–110.

[32] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, A. De Lucia, Automatic test case generation: What if test code quality matters?, in: Proceedings of the 25th International Symposium on Software Testing and Analysis, ACM, 2016, pp. 130–141.

[33] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, A. De Lucia, The scent of a smell: An extensive comparison between textual and structural smells, IEEE Transactions on Software Engineering.

[34] F. Palomba, A. Zaidman, Does refactoring of test smells induce fixing flaky tests?, in: Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on, IEEE, 2017, pp. 1–12.

[35] F. Palomba, A. Zaidman, The smell of fear: On the relation between test smells and flaky tests, Journal of Empirical Software Engineering (2019) in press.

[36] F. Palomba, M. Zanoni, F. A. Fontana, A. De Lucia, R. Oliveto, Toward a smell-aware bug prediction model, IEEE Transactions on Software Engineering.

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

[38] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, A. De Lucia, Methodbook: Recommending move method refactorings via relational topic models, IEEE Transactions on Software Engineering 40 (7) (2014) 671–694.

[39] R. Marinescu, Detection strategies: metrics-based rules for detecting design flaws, in: 20th IEEE International Conference on Software Maintenance, 2004. Proceedings., 2004, pp. 350–359. doi:10.1109/ICSM.2004.1357820.

[40] N. Moha, Y.-G. Gueheneuc, L. Duchien, A.-F. Le Meur, Decor: A method for the specification and detection of code and design smells, IEEE Transactions on Software Engineering 36 (1) (2010) 20–36.

[41] R. Morales, Z. Soh, F. Khomh, G. Antoniol, F. Chicano, On the use of developers? context for automatic refactoring of software anti-patterns, Journal of Systems and Software 128 (2017) 236–251.

[42] M. J. Munro, Product metrics for automatic identification of "bad smell" design problems in java source-code, in: Proceedings of the 11th IEEE International Software Metrics Symposium, METRICS '05, IEEE Computer Society, Washington, DC, USA, 2005, pp. 15–. doi:10.1109/METRICS.2005.38.

[43] F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, A. Zaidman, A textual-based technique for smell detection, in: Program Comprehension (ICPC), 2016 IEEE 24th International Conference on, IEEE, 2016, pp. 1–10.

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

[45] N. Tsantalis, A. Chatzigeorgiou, Identification of move method refactoring opportunities, IEEE Transactions on Software Engineering 35 (3) (2009) 347–367.

[46] F. Palomba, A. Zaidman, A. De Lucia, Automatic test smell detection using information retrieval techniques, in: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2018, pp. 311–322.

[47] F. Palomba, D. A. Tamburri, F. Arcelli Fontana, R. Oliveto, A. Zaidman, A. Serebrenik, Beyond technical aspects: How do community smells influence the intensity of code smells?, IEEE Transactions on Software Engineering.

[48] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, E. Figueiredo, A review-based comparative study of bad smell detection tools, in: Proceedings of the 20th International Conference on Evaluation and Assessment in

Software Engineering, EASE '16, ACM, New York, NY, USA, 2016, pp. 18:1–18:12. doi:10.1145/2915970.2915984.

[49] M. Zhang, T. Hall, N. Baddoo, Code bad smells: A review of current knowledge, J. Softw. Maint. Evol. 23 (3) (2011) 179–202. doi:10.1002/smr.521.

[50] F. A. Fontana, J. Dietrich, B. Walter, A. Yamashita, M. Zanoni, Antipattern and code smell false positives: Preliminary conceptualization and classification, in: Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on, Vol. 1, IEEE, 2016, pp. 609–613.

[51] M. V. Mäntylä, C. Lassenius, Subjective evaluation of software evolvability using code smells: An empirical study, Empirical Software Engineering 11 (3) (2006) 395–431.

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

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

[54] E. Alpaydin, Introduction to machine learning, MIT press, 2014.

[55] R. D. Wangberg, A. Yamashita, A literature review on code smells and refactoring, Master's thesis, Department of Informatics, University of Oslo (2010).

[56] G. Vale, E. Figueiredo, R. Abílio, H. Costa, Bad smells in software product lines: A systematic review, in: Proceedings of the 2014 Eighth Brazilian Symposium on Software Components, Architectures and Reuse, SBCARS '14, IEEE Computer Society, Washington, DC, USA, 2014, pp. 84–94. doi:10.1109/SBCARS.2014.21.

[57] G. Rasool, Z. Arshad, A review of code smell mining techniques, J. Softw. Evol. Process 27 (11) (2015) 867–895. doi:10.1002/smr.1737. URL http://dx.doi.org/10.1002/smr.1737

[58] A. Panichella, R. Oliveto, A. D. Lucia, Cross-project defect prediction models: L'union fait la force, in: 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE), 2014, pp. 164–173. doi:10.1109/CSMR-WCRE.2014.6747166.

[59] B. Ghotra, S. McIntosh, A. E. Hassan, Revisiting the impact of classification techniques on the performance of defect prediction models, in: Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE '15, IEEE Press, Piscataway, NJ, USA, 2015, pp. 789–800.

[60] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, K. Matsumoto, An empirical comparison of model validation techniques for defect prediction models, IEEE Transactions on Software Engineering 43 (1) (2017) 1–18.

[61] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, B. Murphy, Cross-project defect prediction: A large scale experiment on data vs. domain vs. process, in: Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC/FSE '09, ACM, New York, NY, USA, 2009, pp. 91–100. doi:10.1145/1595696.1595713.

[62] B. Kitchenham, S. Charters, Guidelines for performing Systematic Literature Reviews in Software Engineering, School of Computer Science and Mathematics, Keele University, UK (July 2007).

[63] S. U. Khan, Intercultural challenges in offshore software development outsourcing relationships: an exploratory study using a systematic literature review, IET Software 8 (2014) 161–173(12).

[64] A. Tarhan, G. Giray, 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, EASE'17, ACM, New York, NY, USA, 2017, pp. 2–11. doi:10.1145/3084226.3084261.

[65] M. Gasparic, A. Janes, What recommendation systems for software engineering recommend: A systematic literature review, Journal of Systems and Software 113 (Supplement C) (2016) 101 – 113. doi:https://doi.org/10.1016/j.jss.2015.11.036.

[66] J. Vilela, J. Castro, L. E. G. Martins, T. Gorschek, Integration between requirements engineering and safety analysis: A systematic literature review, Journal of Systems and Software 125 (Supplement C) (2017) 68 – 92. doi:https://doi.org/10.1016/j.jss.2016.11.031.

[67] C. Wohlin, Guidelines for snowballing in systematic literature studies and a replication in software engineering, in: Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering, EASE '14, ACM, New York, NY, USA, 2014, pp. 38:1–38:10. doi:10.1145/2601248.2601268.

[68] K. Krippendorff, Content Analysis: An Introduction to Its Methodology (second edition), Sage Publications, 2004.

[69] J.-Y. Antoine, J. Villaneau, A. Lefeuvre, Weighted krippendorff's alpha is a more reliable metrics for multi-coders ordinal annotations: experimental studies on emotion, opinion and coreference annotation., in: G. Bouma, Y. Parmentier (Eds.), Proceedings of the 14th Conference of the European Chapter of the Association for Computational Linguistics (EACL), The Association for Computer Linguistics, 2014, pp. 550–559.
URL `http://dblp.uni-trier.de/db/conf/eacl/eacl2014.html#AntoineVL14`

[70] W. H. Brown, R. C. Malveau, H. W. McCormick, T. J. Mowbray, AntiPatterns: refactoring software, architectures, and projects in crisis, John Wiley & Sons, Inc., 1998.

[71] R. S. Pressman, Software engineering: a practitioner's approach, Palgrave Macmillan, 2005.

[72] A. S. Nuñez-Varela, H. G. Pérez-Gonzalez, F. E. Martínez-Perez, C. Soubervielle-Montalvo, Source code metrics: A systematic mapping study, Journal of Systems and Software 128 (2017) 164–197.

[73] W. Li, S. Henry, Object-oriented metrics that predict maintainability, Journal of systems and software 23 (2) (1993) 111–122.

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

[75] A. Perez, R. Abreu, Framing program comprehension as fault localization, Journal of Software: Evolution and Process 28 (10) (2016) 840–862.

[76] J. Padilha, J. Pereira, E. Figueiredo, J. Almeida, A. Garcia, C. Santanna, On the effectiveness of concern metrics to detect code smells: an empirical study, in: International Conference on Advanced Information Systems Engineering, Springer, 2014, pp. 656–671.

[77] E. Figueiredo, C. Sant'Anna, A. Garcia, T. T. Bartolomei, W. Cazzola, A. Marchetto, On the maintainability of aspect-oriented software: A concern-oriented measurement framework, in: Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on, IEEE, 2008, pp. 183–192.

[78] E. Figueiredo, A. Garcia, M. Maia, G. Ferreira, C. Nunes, J. Whittle, On the impact of crosscutting concern projection on code measurement, in: Proceedings of the tenth international conference on Aspect-oriented software development, ACM, 2011, pp. 81–92.

[79] G. Catolino, F. Palomba, A. De Lucia, F. Ferrucci, A. Zaidman, Developer-related factors in change prediction: an empirical assessment, in: Proceedings of the 25th International Conference on Program Comprehension, IEEE Press, 2017, pp. 186–195.

[80] G. Catolino, F. Palomba, A. De Lucia, F. Ferrucci, A. Zaidman, Enhancing change prediction models using developer-related factors, Journal of Systems and Software.

[81] G. Catolino, F. Ferrucci, Ensemble techniques for software change prediction: A preliminary investigation, in: Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE), 2018 IEEE Workshop on, IEEE, 2018, pp. 25–30.

[82] D. Di Nucci, F. Palomba, G. De Rosa, G. Bavota, R. Oliveto, A. De Lucia, A developer centered bug prediction model, IEEE Transactions on Software Engineering.

[83] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, O. Strollo, When does a refactoring induce bugs? an empirical study, in: Source Code Analysis and Manipulation (SCAM), 2012 IEEE 12th International Working Conference on, IEEE, 2012, pp. 104–113.

[84] S. R. Safavian, D. Landgrebe, A survey of decision tree classifier methodology, IEEE transactions on systems, man, and cybernetics 21 (3) (1991) 660–674.

[85] C. Cortes, V. Vapnik, Support vector machine, Machine learning 20 (3) (1995) 273–297.

[86] A. Corazza, S. Di Martino, F. Ferrucci, C. Gravino, F. Sarro, E. Mendes, Using tabu search to configure support vector regression for effort estimation, Empirical Software Engineering 18 (3) (2013) 506–546.

[87] T. Hall, S. Beecham, D. Bowes, D. Gray, S. Counsell, A systematic literature review on fault prediction performance in software engineering, IEEE Transactions on Software Engineering 38 (6) (2012) 1276–1304.

[88] D. D. Nucci, F. Palomba, R. Oliveto, A. D. Lucia, Dynamic selection of classifiers in bug prediction: An adaptive method, IEEE Transactions on Emerging Topics in Computational Intelligence 1 (3) (2017) 202–212. doi:10.1109/TETCI.2017.2699224.

[89] D. Di Nucci, F. Palomba, A. De Lucia, Evaluating the adaptive selection of classifiers for cross-project bug prediction.

[90] S. W. Thomas, M. Nagappan, D. Blostein, A. E. Hassan, The impact of classifier configuration and classifier combination on bug localization, IEEE Transactions on Software Engineering 39 (10) (2013) 1427–1443.

[91] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, K. Matsumoto, The impact of automated parameter optimization on defect prediction models, IEEE Transactions on Software Engineering PP (99) (2018) 1–1. doi:10.1109/TSE.2018.2794977.

[92] C.-W. Hsu, C.-C. Chang, C.-J. Lin, et al., A practical guide to support vector classification, Tech. rep., Department of Computer Science, National Taiwan University (7 2003).

[93] P. He, B. Li, X. Liu, J. Chen, Y. Ma, An empirical study on software defect prediction with a simplified metric set, Information and Software Technology 59 (C) (2015) 170–190.

[94] S. Watanabe, H. Kaiya, K. Kaijiri, Adapting a fault prediction model to allow inter languagereuse, in: Proceedings of the 4th International Workshop on Predictor Models in Software Engineering, ACM, 2008, pp. 19–24.

[95] P. A. Devijver, J. Kittler, Pattern recognition: A statistical approach, Prentice hall, 1982.

[96] T. Hall, S. Beecham, D. Bowes, D. Gray, S. Counsell, Developing fault-prediction models: What the research can show industry, IEEE Software 28 (6) (2011) 96–99.

[97] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, A. De Lucia, Detecting code smells using machine learning techniques: are we there yet?, in: 25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER2018): REproducibility Studies and NEgative Results (RENE) Track, Institute of Electrical and Electronics Engineers (IEEE), 2018.

[98] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, J. Noble, The qualitas corpus: A curated collection of java code for empirical studies, in: Software Engineering Conference (APSEC), 2010 17th Asia Pacific, IEEE, 2010, pp. 336–345.

[99] F. Palomba, D. Di Nucci, M. Tufano, G. Bavota, R. Oliveto, D. Poshyvanyk, A. De Lucia, Landfill: An open dataset of code smells with public evaluation, in: Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on, IEEE, 2015, pp. 482–485.

[100] H. Cooper, L. V. Hedges, J. C. Valentine, The handbook of research synthesis and meta-analysis, Russell Sage Foundation, 2009.

[101] R. E. Slavin, Best-evidence synthesis: An alternative to meta-analytic and traditional reviews, Educational researcher 15 (9) (1986) 5–11.

[102] A. P. Field, Meta-analysis of correlation coefficients: a monte carlo comparison of fixed-and random-effects methods., Psychological methods 6 (2) (2001) 161.

[103] M. Borenstein, L. V. Hedges, J. P. Higgins, H. R. Rothstein, A basic introduction to fixed-effect and random-effects models for meta-analysis, Research synthesis methods 1 (2) (2010) 97–111.

[104] B. A. Kitchenham, D. Budgen, P. Brereton, Evidence-based software engineering and systematic reviews, Vol. 4, CRC Press, 2015.

[105] S. Hosseini, B. Turhan, D. Gunarathna, A systematic literature review and meta-analysis on cross project defect prediction, IEEE Transactions on Software Engineering (2018) 1–1doi:10.1109/TSE.2017.2770124.

[106] Y. Rafique, V. B. Mišić, The effects of test-driven development on external quality and productivity: A meta-analysis, IEEE Transactions on Software Engineering 39 (6) (2013) 835–856.

[107] J. E. Hannay, T. Dybå, E. Arisholm, D. I. Sjøberg, The effectiveness of pair programming: A meta-analysis, Information and Software Technology 51 (7) (2009) 1110–1122.

[108] M. W. Lipsey, D. B. Wilson, Practical meta-analysis., Sage Publications, Inc, 2001.

[109] S. Greenland, M. P. Longnecker, Methods for trend estimation from summarized dose-response data, with applications to meta-analysis, American journal of epidemiology 135 (11) (1992) 1301–1309.

[110] E. Tacconelli, Systematic reviews: Crd's guidance for undertaking reviews in health care, The Lancet Infectious Diseases 10 (4) (2010) 226.

[111] J. M. Bland, D. G. Altman, The odds ratio, Bmj 320 (7247) (2000) 1468.

[112] B. Turhan, T. Menzies, A. B. Bener, J. Di Stefano, On the relative value of cross-company and within-company data for defect prediction, Empirical Software Engineering 14 (5) (2009) 540–578.

[113] F. Peters, T. Menzies, A. Marcus, Better cross company defect prediction, in: Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on, IEEE, 2013, pp. 409–418.

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

[115] C. Simons, J. Singer, D. R. White, Search-based refactoring: Metrics are not enough, in: International Symposium on Search Based Software Engineering, Springer, 2015, pp. 47–61.

[116] I. Candela, G. Bavota, B. Russo, R. Oliveto, Using cohesion and coupling for software remodularization: Is it enough?, ACM Transactions on Software Engineering and Methodology (TOSEM) 25 (3) (2016) 24.

[117] V. Kovalenko, F. Palomba, A. Bacchelli, Mining file histories: should we consider branches?, in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ACM, 2018, pp. 202–213.

[118] F. A. Fontana, V. Ferme, M. Zanoni, R. Roveda, Towards a prioritization of code debt: A code smell intensity index, in: Managing Technical Debt (MTD), 2015 IEEE 7th International Workshop on, IEEE, 2015, pp. 16–24.

[119] D. Di Nucci, F. Palomba, R. Oliveto, A. De Lucia, Dynamic selection of classifiers in bug prediction: An adaptive method, IEEE Transactions on Emerging Topics in Computational Intelligence 1 (3) (2017) 202–212.

AppendixA. List of journals and conferences manually searched

Table A.10: List of journals manually searched

S.No	Journal name
1	IEEE Transactions on Software Engineering
2	ACM Transactions on Software Engineering and Methodology
3	Empirical Software Engineering
4	Software Quality Journal
5	Journal of Systems and Software (JSS)
6	Information & Software Technology (IST)
7	Journal of Automated Software Engineering
8	Software-Practice and Experience (SPE)
9	Journal of Software: Evolution and Process
10	Journal of Software Maintenance and Evolution: Research and Practice
11	IEEE Software

Table A.11: List of Conferences manually searched

S.No	**Conference name**
1	IEEE/ACM International Conference on Software Engineering (ICSE) 2007-2017
2	ACM/SIGSOFT Symposium on the Foundations of Software Engineering (FSE) 2016, 2014, 2012, 2010, 2011, 2008
3	Joint Meeting of the European Software Engineering Conference and the ACM/SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE) 2007, 2009, 2013
4	IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007-2016
5	European Conference on Object Oriented Programming (ECOOP) 2007-2017
6	Fundamental Approaches to Software Engineering - International Conference (FASE), Held as Part of the European Joint Conferences on Theory and Practice of Software, 2007-2017
7	IEEE International Conference on Software Maintenance (ICSM), 2007-2013
8	International Conference on Software Maintenance and Evolution (ICSME), 2014-2016
9	International Conference on Mining Software Repositories (MSR), 2008-2011, 2016-2017
10	ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 2007-2017
11	International Conference on Evaluation and Assessment in Software Engineering (EASE), 2007-2017
12	Software Engineering and Knowledge Engineering (SEKE), 2007-2017
13	International Conference on Program Comprehension (ICPC), 2007-2017
14	IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2015-2017
15	IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE) 2014
16	Working Conference on Reverse Engineering (WCRE), 2007-2013
17	European Conference on Software Maintenance and Reengineering (CSMR), 2007-2013
18	IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), 2007-2016
19	ACM Object-oriented Programming, Systems, Languages, and Applications (OOPSLA), 2007-2017
20	International Conference on Advanced Information Systems Engineering (CAiSE), 2007-2018
21	IEEE International Conference on Software Reuse (ICSR), 2006-2018

AppendixB. List of all the metrics and their definitions used in the literature on code smells prediction

Table B.12: Metrics names along with their definitions

Name	Definition	Name	Definition
ACAIC	Ancestor Class-Attribute Import Coupling	ACMIC	Ancestors Class-Method Import Coupling
AID	Average Inheritance Depth of an entity	ANA	Average number of entities from which an entity inherits information wrt. to the total number of entities in a model
CAM	Relatedness among methods of an entity based on the parameter list of its methods	CBO	Coupling Between Objects of one entity
CBOingoing	Coupling Between Objects of one entity (in-going coupling only, towards the entity)	CBOoutgoing	Coupling Between Objects of one entity (out-going coupling only, from the entity)
CIS	Number of public methods in a class	CLD	Class to Leaf Depth of an entity
cohesionAttributes	Percentage of fields (declared and inherited) used by the declared methods of one entity wrt. all its fields	connectivity	Number of couples of methods that use each other
CP	Number of packages that depend on the package containing the entity	DAM	Ratio of the number of private (and protected) fields wrt. the total number of fields declared in an entity
DCAEC	Descendants Class-Attribute Export Coupling of one entity	DCCdesign	Number of classes that a class is directly related to (by attribute and parameter declarations)
FANIN		DCMEC	Descendants Class-Method Export Coupling of one entity
DIT	Depth of Inheritance Tree of an entity	DSC	Number of top-level entities in a model
EIC	Number of inheritance relationships in which super-entities are in external packages	EIP	Number of inheritance relationships where the super-entity is in the package containing the entity and the sub-entites is in another package
FanOut	Number of methods and fields used by one entity	ICHClass	Complexity of an entity as the sum of the complexities of its declared and inherited methods
IR	Number of calls from the methods of an entity to the methods and fields declared in its super-entities	LCOM1	Lack of COhesion in Methods of an entity
LCOM2	Lack of COhesion in Methods of an entity	LCOM5	Lack of COhesion in Methods of an entity
LOC	Sum of the numbers of lines of code in the methods of an entity	McCabe	Sum of the cyclomatic complexities of the operations of an entity
MFA	Ratio of the number of methods inherited by an entity wrt. the number of methods accessible by member methods of the entity	MLOCsum	Sum of the numbers of lines of code in the methods of an entity. Same as LOC
MOA	Number of data declarations whose types are user-defined entities	NAD	Number of Attributes Declared by an entity
NADextended	Number of Attributes Declared by an entity and in its member entities	NCM	Number of Changed Methods of an entity wrt. its hierarchy
NCP	Number of Classes per Package, i.e., the number of classes within a package	NMA	Number of Methods Added by an entity with respect to its hierarchy
NMD	Number of Methods Declared by an entity	NMDextended	Number of Methods Declared by an entity and its member entities
NMI	Number of Methods Inherited by an entity. Constructors or not considered as method, they are not counted in the result of the metric	NMO	Number of Methods Overridden by an entity
NOA	Number Of Ancestors of an entity	NOC	Number Of Children of an entity
NOD	Number of descendants of an entity	NOF	Number Of Fields declared by an entity
NOH	Number Of Hierarchies in a model	NOM	Number Of Methods declared by an entity
NOP	Number Of Parents of an entity	NOParam	Number of parameters of the methods of an entity
NOPM	Number Of Polymorphic Methods in an entity wrt. its hierarchy	NOTC	Number of invocations of JUnit assert methods that occur in the code of a test case
NOTI	Number Of Transitive Invocation among methods of a class. See the Law of Demeter for a definition	NPrM	Number protected members of an entity
NOII	Number of Implemented Interfaces	PIIR	Number of inheritance relationships existing between entities in the package containing an entity
PP	Number of provider packages of the package containing an entity	REIP	EIP divided by the sum of PIIR and EIP

Table B.12: Metrics names along with their definitions (continue..)

Name	Definition	Name	Definition
RFC	Response for class: number of methods of an entity and of methods of other entities that are invoked by the methods of the entity	RFCextended	Response for class: number of methods of an entity and of methods of other entities that are invoked by the methods of the entity plus number of methods declared by that entity
RFP	Number of references from entities belonging to other packages to entities belonging to the package containing an entity	RPII	PIIR divided by the sum of PIIR and EIP
RRFP	RFP divided by the sum of RFP and the number of internal class references	RRTP	RTP divided by the sum of RTP and the number of internal class references
RTP	Number of references from entities in the package containing an entity to entities in other packages	SIX	Specialisation IndeX of an entity
		TLOC	Number of lines of code of all the methods of an entity. Same as LOC
VGSum	Sum of the cyclomatic complexities of the operations of an entity. Same as McCabe	WMC1	Weight of an entity considering the complexity of each of its method as being 1. (Default constructors are considered even if not explicitly declared)
WMCinvocations	Weight of an entity considering the complexity of each of its method as being the numbers of invocations that they perform. (Default constructors are considered even if not explicitly declared).	WMCmccabe	Weight of an entity considering the complexity of each of its method as being their McCabe cyclomatic complexity. (Default constructors are considered even if not explicitly declared).
LOCNAMM	Lines of Code Without Accessor or Mutator Methods	NOPK	Number of Packages
NOCS	Number of Classes	NOMNAMM	Number of Not Accessor or Mutator Methods
CYCLO	Cyclomatic Complexity	WMC	Weighted Methods Count
WMCNAMM	Weighted Methods Count of Not Accessor or Mutator Methods	AMW	Average Methods Weight
NODA	Number of default Attributes	NOPVA	Number of Private Attributes
NOPRA	Number of Protected Attributes	NOFA	Number of Final Attributes
NOFSA	Number of Final and Static Attributes	NOFNSA	Number of Final and non - Static Attributes
NONFNSA	Number of not Final and non - Static Attributes	NOSA	Number of Static Attributes
NONFSA	Number of non - Final and Static Attributes	NOABM	Number of Abstract Methods
NONCM	Number of non - Constructor Methods	NOFM	Number of Final Methods
NOFNSM	Number of Final and non - Static Methods	NOFSM	Number of Final and Static Methods
NONFNABM	Number of non - final and non - abstract Methods	NONFNSM	Number of Final and non-Static Methods
NONFSM	Number of non - Final and Static Methods	NODM	Number of default Methods
NOPM	Number of Private Methods	NOPRM	Number of Protected Methods
NOPLM	Number of Public Methods	NONAM	Number of non-Accessors Methods
NOSM	Number of Static Methods	NOCM	Number of Constructor Methods
AMWNAMM	Average Methods Weight of Not Accessor or Mutator Methods	MAXNESTING	Maximum Nesting Level
CLNAMM	Called Local Not Accessor or Mutator Methods	NOAV	Number of Accessed Variables
ATLD	Access to Local Data	NOLV	Number of Local Variable
ATFD	Access to Foreign Data	FDP	Foreign Data Providers
CFNAMM	Called Foreign Not Accessor or Mutator Methods	CINT	Coupling Intensity
MaMCL	Maximum Message Chain Length	MeMCL	Mean Message Chain Length
NMCS	Number of Message Chain Statements	CC	Changing Classes
CM	Changing Methods	NOAM	Number of Accessor Methods
NOAM	Number of Accessor Methods	NOPA (NOAP)	Number of Public Attribute
LAA	Locality of Attribute Accesses	NOI	Number of Interfaces
LOC_1	Customized LOC	RFC_New	Customized RFC
WMC_New	Customized WMC	DIT_1	Customized DIT
D_APPEAR	Appearance in Diagrams	ABSTR_R	Number of Abstractions
ASSOC_R	Number of Associations	DEPEND_R	Number of Dependencies
C_PARAM	Number of times class is used as parameter type	NA	Number of Attributes
NC	Number of Classes	NM	Number of Members
ACT	Number of Actors	COMP	Number of Components
NS	Number of Namespaces	WAC	Weighted Attributes per Class
WMC	Weighted Methods per Class	NP	Number of Parameters
AHF	Attribute Hiding Factor	AIF	Attribute Inheritance Factor
CF	Coupling Factor	MHF	Method Hiding Factor
MIF	Method Inheritance Factor	PF	Polymorphism Factor

Table B.12: Metrics names along with their definitions (continue..)

Name	Definition	Name	Definition
NAI	Number of Inherited Attributes	NO	Number of Operation
NOLV	No of Local Variables of a Method		

AppendixC. Independent variables used across different studies with metrics set label

Table C.13: Performance of code smell prediction models based on the different combination of independent variables adopted by the primary studies.

Studies	Independent Variables	Metrics Set Label
[S01], [S10], [S13]	LOC, LOCNAMM, NOM, NOPK, NOCS, NOMNAMM, NOA, CYCLO, WMC, WMCNAMM, AMWNAMM, AMW, MAXNESTING, WOC, CLNAMM, NOP, NOAV, ATLD, NOLV, FANOUT, FANIN, ATFD, FDP, RFC, CBO, CFNAMM, CINT, CDISP, MaMCL, MeMCL, NMCS, CC, CM, LAA, NOAM, NOPA, DIT, NOI, NOC, NMO, NIM, NOII, LCOM5, TCC, NODA , NOPVA, NOPRA, NOFA, NOFSA, NOFNSA, NONFNSA, NOSA, NONFSA, NOABM, NOCM, NONCM, NOFM, NOFNSM, NOFSM, NONFNABM, NONFNSM, NONFSM, NODM, NOPM, NOPRM, NOPLM, NONAM, NOSM	A
[S06], [S08]	LOC, NAD, NADextended, NMA, NMD, NMDextended, NOM McCabe, NOParam, WMC1, WMC, CIS, ACAIC, ACMIC, CAM, CBO, CBOingoing, CBOoutgoing, connectivity, DCAEC, DCMEC, IR, NCM, NOTI, RFC, DSC, DCC, DAM, NPrM, AID, ANA, CLD, DIT, ICHClass, MFA, NMI (NIM), NMO, NOA, NOC, NOD, NOH, NOP, NOPM, cohesionAttributes, LCOM1, LCOM2, LCOM5, MOA, SIX, USELESS	B
[S09]	LOC, LOC_1, MLOCsum, NAD, NADExtended, NMA, NMD, NMDExtended, NOM, WMC, McCabe, NOF, NOP, NOParam, Vgsum, WMC1, WMC_New, CBO, RFC, CA, CE, CAM, ACAIC, ACMIC, DCAEC, DCC, DCMEC, IR, NCM, NOTI, RFC_New, connectivity, DSC, DAM, DIT, NOC, MFA, AID, CLD, DIT_1, ICHClass, NMI, NMO, NOA, NOC_1, NOD, NOH, NOPM, LCOM, LCOM3, LCOM1, LCOM2, LCOM5, cohesionAttributes, NPM, MOA, IC, CBM, AMC, NOTC, SIX	C
[S04]	NAD, NADextended, NCP, NMA, NMD, NMDextended, NOM, PP, CIS, McCabe, NOParam, WMC1, WMCinvocations, WMCmccabe, ACAIC, ACMIC, CAM, CBO, CBOingoing, CBOoutgoing, connectivity, CP, DCAEC, DCCdesign, DCMEC, FanOut, NCM, RFP, RTP, DSC, DAM AID, ANA, CLD, DIT, EIC, EIP, ICHClass, MFA, NMI, NMO, NOA, NOC, NOD, NOH, NOP, NOPM, PIIR, cohesionAttributes, LCOM1, LCOM2, LCOM5, MOA, REIP, RPII, RRFP, RRTP, SIX	D
[S05]	NOA, NOCS, NOMembers, NOM, ACT, COMP, NS, WAC, WMA, NOP, CBO, RFC, AHF, AIF, CF, MHF, MIF, PF, DIT, NOC, NAI, NMI, C_PARAM, D_APPEAR, ABSTR_R, ASSOC_R, DEPEND_R	E
[S07]	LOC, Number of Invocations, Number of Library Invocations, Number of Local Invocations, Number of Other Invocations, Number of Field Accesses, Number of Invocations, Number of Library Invocations, Number of Local Invocations, Number of Other Invocations, Number of Field Accesses, History Features (Existence Time, Number of Changes, Number of Recent Changes, File Existence Time, Number of File Changes, Number of Recent File Changes), Destination Features (Whether it is a Local Clone, Fine Name Similarity, Masked File Name Similarity, Method Name Similarity, Sum of Parameter Similarities, Maximal Parameter Similarity, Difference on Only Postfix Number)	F
[S14], [S15]	Tokenized source code e.g. terms of the class and programming constructs used in the class etc.	G