Neurocomputing 565 (2024) 127014

Contents lists available at ScienceDirect

Neurocomputing

journal homepage: www.elsevier.com/locate/neucom

Survey paper

Code smell detection based on supervised learning models: A survey

Yang Zhang [a], Chuyan Ge [b,*], Haiyang Liu [a], Kun Zheng [a]

[a] School of Information Science and Engineering, Hebei University of Science and Technology, Shijiazhuang 050000, China
[b] Faculty of Computing, Harbin Institute of Technology, Harbin 150006, China

ARTICLE INFO

Communicated by Zidong Wang

Keywords:
Code smell
Supervised learning
Feature selection
Dataset
Data balancing

ABSTRACT

Supervised learning-based code smell detection has become one of the dominant approaches to identify code smell. Existing works optimize the process of code smell detection from multiple aspects, such as high-quality dataset, feature selection, and model, etc. Although the accuracy is improved continuously, researchers are confused about what model are the most suitable ones to detect code smell when considering dataset construction and feature selection. Furthermore, existing surveys for code smell mainly analyze the impact of code smell, categorize the concerns of code smell, and repair code smell. There is a lack of systematic analysis and classification of code smell detection based on supervised learning. To this end, we collect 86 papers of code smell detection based on supervised learning ranging from January 2010 to April 2023. A total of 7 research questions is empirically evaluated from different aspects, such as datasets construction, data pre-processing, feature selection, and model training, etc. We conclude that existing works suffer from issues such as sample imbalance, different attention to types of code smell, and limited feature selection. Finally, we suggest possible future research directions.

1. Introduction

Code smell is a kind of code structure that violates design principles and has a negative impact on code quality, which is a symptom of poor software design and bad coding habits. Unlike bugs in source code, a program with code smell can run smoothly. Although it does not have a short-term impact on the run of a program, the long-term impact of code smell tends to reduce the quality of software and make it difficult to understand and maintain.

Since the concept of code smell has been proposed in the late 1990s [1], many researches have been conducted continuously by both academia and industrial section. Most works focus on the severity [2,3], correlation [4,5], detection [6-8] and repairation [9,10] of code smell. Among these works, more than 90 % are subjective to code smell detection [11], demonstrating that code smell detection has been always a hot topic in the field of software evolution.

To identify the code smell, researchers leverage both manual and automatic detection approaches. Early detection for code smell are conducted manually. However, this approach not only requires the expertise, but also takes time and effort. Furthermore, the detection accuracy are unsatisfying. Sjoberg et al. [12] illustrated that the manual detection of code smell was impractical for large industrial applications. Therefore, most researchers prefer automated approaches to identify code smell quickly and accurately. Early automatic detection tools mostly rely on code metrics and heuristic rules [13]. To identify code smell, it usually selects metrics manually and sets threshold to judge whether or not code structures can meet these metrics and rules. Since the first automatic detection tool for code smell was introduced, many famous tools (such as DECOR [13], JDedorant [14], iPlasma [2], etc.) have been proposed to improve the accuracy of code smell detection.

Although many automated tools have been developed, there are some shortcomings in these approaches. Firstly, distinct developers use different rules or criteria. As a result, it leads to varying results of the same code smell detection by different detection tools, resulting in a high false positive rate. Secondly, selecting the appropriate metrics or rules requires expertise. The weights of different metrics and rules may vary among code smell. Ignoring those metrics with high weights may decrease the performance of the detection tool. To avoid the limitations of heuristic rule-based approaches, genetic algorithm-based detection [15], semantic-based detection [16], and history-based detection [17] are proposed to identify code smell [18–20].

With the development of artificial intelligence in recent years, some researchers start to leverage machine learning or deep learning models to detect code smell. Supervised learning, semi-supervised learning, and

* Correspondence to: Harbin Institute of Technology, No 92, Xidazhi Street, Harbin, Heilongjiang Province, China.
E-mail address: 15531172710@163.com (C. Ge).

https://doi.org/10.1016/j.neucom.2023.127014
Received 4 May 2023; Received in revised form 1 November 2023; Accepted 6 November 2023
Available online 9 November 2023

unsupervised learning can be employed. Semi-supervised learning is usually performed on a small number of labeled samples and a large number of unlabeled samples. Since the number of unlabeled samples is more than the number of labeled samples and the features of different code smell are different, it is difficult to improve the detection accuracy in the process of model learning. Unsupervised learning is performed on unlabeled samples. Considering that many kinds of code smells have similar characteristics, the detection for these code smells is always ambiguous. Therefore, supervised learning provides an explicit classification for model training and prediction by labeled samples, which has been the mainstream for code smell detection.

To provide an overview of various approaches and recent researches on code smell detection, researchers survey code smell from different aspects. For example, Zhang et al. [21] analyzed the impact of code smell on software evolution and studied code smell from an empirical perspective. Tian et al. [22] investigated the difference in the emphasis on code smell between industry and academia and its impact. Azeem et al. [23,24] systematically analyzed different approaches in code smell detection. Although some surveys already exist, there is few work on code smell detection based on supervised learning.

Conducting a survey on supervised learning-based code smell detection is necessary. First, although existing works have achieved high accuracy, there are numerous supervised learning models and various classification approaches which still need further analysis. Secondly, the existing works on supervised learning approaches utilizes different datasets, different feature selection approaches, and different data balancing approaches which need to be analyzed and summarized. In addition, what model verification and evaluation exist in supervised learning based code smell detection methods? Which kind of verification and evaluation methods are used more often? How effective are they? These questions need to be further clarified. Finally, the problems of supervised learning models in the selection of independent variables and the application of feature selection techniques remain insufficiently and effectively studied.

To address the above problems, this paper focuses on supervised learning-based code smell detection and collects 86 papers published between January 2010 and April 2023. A total of 7 research questions is empirically evaluated from different aspects, such as datasets construction, data pre-processing, feature selection, and model training, etc. We conclude that existing works suffer from issues such as sample imbalance, different attention to types of code smell, and limited feature selection. Finally, we suggest possible future research directions.

This paper is organized as follows. Section 2 introduces the research methodology of this paper. Section 3 presents the framework of supervised learning-based code smell detection and seven research questions. Sections 4–8 answer each research question. Section 9 presents the problems of code smell detection based on supervised learning and suggests possible research directions. Related works are examined in Section 10 before conclusions are drawn in Section 11.

2. Methodology

This section describes the search strategy for paper collection and presents the search procedure and results.

2.1. Search strategy

We designed a series of search strategies to collect all papers related to the topic of our paper. Our search strategies include defining the search string, the library of search resources, inclusion, and exclusion criteria, etc.

2.1.1. Identifying the search words

To find relevant search words, we adopt the following steps.

(1) The main words are defined as keywords through the research questions.
(2) For all major words, we find alternative spellings and synonyms.
(3) Validating the keywords in relevant papers.
(4) The *Boolean* operators OR or AND are used for concatenation. We use the OR operator to concatenate alternative spellings or synonyms, and the AND operator for concatenating the main terms and keywords.
(5) Integrating the keywords into search strings. The following search string is applied: ((“code smell” OR “code bad smell” OR “bad smell” OR “anti-patterns” OR “antipattern” OR “design defect” OR “design smell ”) AND (“software” OR “software engineering”)) AND (“supervised learning” OR “deep learning” OR “machine learning”).

2.1.2. Electronic data sources

Considering the influence of the literature databases, the fields of the included papers, and the authority in the field of computing, we select the following sources to search 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)
- Engineering Village (https://www.engineeringvillage.com)
- CNKI (https://www.cnki.net/)
- Wanfang Database WFPD (https://www.wanfangdata.com.cn/index.html)
- China Science and Technology Journal Database (https://www.cqvip.com/)

These resources are representative in the field of software engineering which contains journal and conference papers related to our research questions and they are widely used in many literature reviews [23].

2.1.3. Inclusion and exclusion criteria

Papers satisfying the following constraints are included in this study.

(1) Papers written in English or Chinese and related to code smell detection.
(2) Papers that improve the performance of code smell detection in technical aspects.
(3) Papers related to code smell detection using supervised learning.

Papers that satisfy the following constraints are excluded from the study.

(1) Papers that only refer to code smell, without further discussion.
(2) Papers are written in other languages except for Chinese or English.
(3) Papers without any content.
(4) Papers that are not in the field of software engineering.
(5) Papers that do not use supervised learning.
(6) Review papers.

2.2. Searching process

To search for relevant papers, we follow four steps.

(1) The research papers are collected by searching the resource library in 2.1.2 using the search string mentioned in Section 2.1.1. After removing duplicates (similar papers published by the same authors in different journals or conferences), a total of 105 papers related to the study content are collected.

(2) After reading the titles, abstracts, and keywords, irrelevant papers are filtered from the entire list of search sources according to the exclusion criteria in 2.1.3. In this step, we excluded 16 papers.
(3) We perform a "snowballing" [25] process to find papers that might have been missed. By reading the titles, abstracts, and keywords of the newly found papers, and applying the inclusion/exclusion criteria, we added 2 papers in this step.
(4) After creating the final set of papers, we conduct a quality assessment to ensure that all papers have the information we needed to answer our research questions. In this step, we exclude 5 papers.

2.3. Search results

After the above search process, we collect 86 papers on supervised learning based code smell detection, including 77 papers in English and 9 papers in Chinese. A total of 66 papers leverages machine learning models and 20 papers employ deep learning models. For the reproducibility of our research, all materials of this survey are available at https://github.com/Gechuyan/Code-Smell-Detection-Survey.

Fig. 1 shows the distribution of related literature in terms of years. It can be seen that machine learning has been used to detect code smell since 2010. Since 2017, researchers have started to publish work on deep learning based code smell detection. Starting from this year, the number of related papers has shown a trend of increasing year by year. The highest number of published papers till now is in 2022. Almost 75% of the relevant studies are published in 2019 and later. The number of papers on supervised learning based code smell detection has maintained more than 12 papers per year since 2019. The data suggest that supervised learning based code smell detection has received increasing attention in recent years.

3. Code smell detection framework and research questions

This section starts with a summary of the framework of supervised learning-based code smell detection approaches and then presents seven research questions.

3.1. Detection framework

The framework of supervised learning based code smell detection is shown in Fig. 2. The primary problem of supervised learning-based code smell detection approaches is to build a dataset. In order to improve the quality of the dataset, researchers often select large-scale real-world applications. Various features are extracted at different granularity (classes, methods, statement blocks, instructions) and different categories (syntax and semantics) are extracted from the source code. The dataset contains a large number of samples among which the samples are labelled through the pre-defined rules. Data pre-processing techniques such as data cleaning, integration, transformation, and augmentation are commonly employed to make a sample of data easier for model learning. Since most large real-world applications are designed carefully with less code smell in the source code, there may be an imbalance of positive and negative samples in the constructed dataset. To solve this problem, researchers use methods such as oversampling, undersampling, and refactoring [26] generation to adjust the number of positive and negative samples in the dataset in order to ensure data balance. In the realm of supervised learning-based code smell detection, a pivotal challenge lies in the development of robust machine learning models or deep neural network models that can effectively classify code smells. This classification task can take various forms, including binary classification, which distinguishes between code with and without smells; severity classification, which categorizes code smells based on their impact or severity; and multiclass classification, which detects a wide range of code smell. Model evaluation is a critical aspect of the process. Researchers assess the performance of the trained models using various strategies like K-fold cross-validation, random sub-sampling, and percentage splitting. Evaluation metrics such as accuracy, precision, recall, and F-measure are used to gauge how well the models perform.

3.2. Research questions

In this paper, we propose 7 research questions (RQ) by considering different aspects of supervised learning based code smell detection. RQ1-RQ7 are analyzed from different perspectives based on existing works.

RQ1: How much attention is paid to different code smell in supervised learning approaches? How many types of code smell can be detected simultaneously?

RQ2: What kind of datasets are used for supervised learning model training?

RQ3: What pre-processing are employed for different types of data in supervised learning methods?

RQ4: How do existing approaches perform feature selection before model training?

RQ5: How do existing works ensure data balance in the datasets in supervised learning methods?

RQ6: What machine learning and deep learning models are used?

RQ7: What are the differences in the validation strategies and evaluation of models in supervised learning based code smell detection?

RQ1 focuses on the degree of attention on the detection of different code smell in supervised learning, and provides statistics in terms of the number of times detected and the number of detected code smell. RQ2 compares the rules of sample labeling and focuses on constructing datasets in existing works. RQ3 provides a systematic analysis of the preprocessing methods for different types of data. RQ4 and RQ5 perform statistical analysis on the feature selection and data imbalance in existing methods, respectively. RQ6 conducts analysis on machine learning and deep learning models. RQ7 focuses on different methods and metrics in the model training and evaluation phases.

4. The hotspot of code smell

To answer RQ1, this section counts the degree of concern for different code smell and gives the number of code smell detected simultaneously.

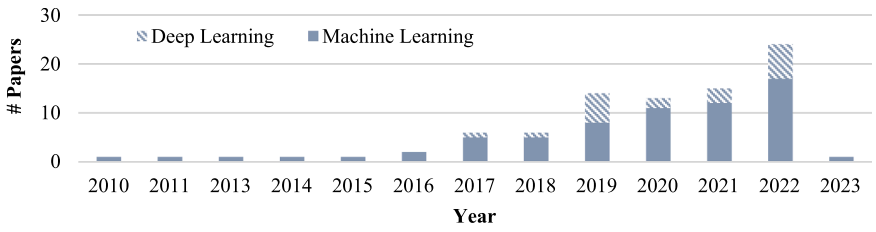

Fig. 1. Number of papers on code smell detection based on supervised learning.

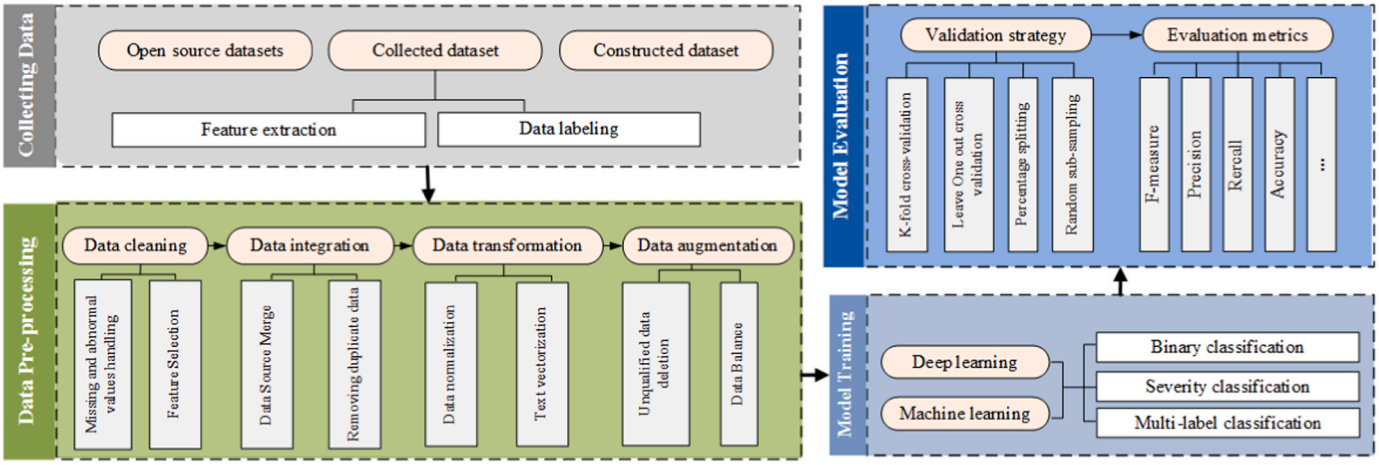

Fig. 2. The framework of supervised learning based code smell detection.

4.1. The frequency of detecting different code smell

Fowler proposed 22 types of code smell such as long parameter lists, feature envy, and data classes [1]. Tian et al. [22] systematically analyzed and categorized 339 papers related to code smell published between 1990 and December 2020, and summarized 38 code smell types, which involve various granularity levels such as class level and method level. Although many code smells have been proposed, we wonder whether these code smells are treated equally in the process of detection? To answer this question, we count the number of papers on each type of code smell detected in the existing work, and the results are shown in Table 1. It should be noted that there may be cases where multiple code smell are detected in one papers. We list the papers that detects code smell separately.

4.2. Code smell detection simultaneously

Does the existing work detect only one kind of code smell? How many kinds of code smell can be detected simultaneously? To answer these questions, we counted the number of code smell detected in each paper. The results are shown in Fig. 3. Obviously, only 16 papers detect only one type of code smell. More than 80% papers detects two or more code smell, indicating that researchers prefer to detect multiple code smell at the same time. We should note that the number that detects two types of code smell is not the largest among all papers. 32% papers are contributed to detecting four kinds of code smell simultaneously, demonstrating the largest number of all papers. By further insighting into these papers, we found that *God Class*, *Feature Envy*, *Data Class*, and *Long Method* code smells are often detected simultaneously [27–30]. The reason is that these four kinds of code smells appear frequently with more obvious features in real-world applications and are easy to identify.

Table 1
The total number of papers examining different code smell.

Code Smell	Literature	Frequency
Feature Envy	[3,7,27–63]	38
Long Method	[3,6,7,27,28–30,32,33–35,37,39,41–47, 49,50,54–56,58,60–70]	37
God Class	[3,7,27–30,32,34,35,39,40–43,47,50,54, 55,58,60–64,66,67,70–79]	36
Data Class	[3,27–30,32,33–36,39,41–43,50,52,54, 55,58,60–63,76,78]	24
Large Class	[3,33,34,37,44,45,53,68,70,80]	10
Long Parameter List	[6,42,44,46,47,51,60,68]	8
Complex Class	[6,64,65,67,68,74,75,79]	8
Lazy Class	[34,44,46,69,74,79]	6
Spaghetti Code	[52,64,67,74,75,79]	6
Brain Class	[3,76,78,81,82]	5
Shotgun Surgery	[45,69,83-85]	5
Refused Parent Bequest	[32,40,51,69,78]	5
Message Chains	[6,46,68,83]	4
Class Data Should Be Private	[46,64,67,68]	4
Speculative Generality	[51,69,74]	3
Parallel Inheritance Hierarchies	[34,45,84]	3
Switch Statements	[42,46,60]	3
Brain Method	[76,81,82]	3
Duplicated Code	[44,69,70]	3
Middle Man	[46,51,79]	3
Divergent Change	[45,84]	2
Inappropriate Intimacy	[51,79]	2

5. The sources of the datasets

To answer RQ2, we count the number of papers that including sources of datasets used for supervised learning. Datasets is one of the important components of supervised learning based code smell detection approaches, and their quality determines the learning quality of the model. Sotto et al. [86] classifies the building of datasets into four categories: collected datasets, open source datasets, constructed datasets, and industry datasets. Fig. 4 shows the sources of datasets in the selected primary studies. It should be noted that no work in the literature we counted used industry datasets, so we only counted the other three being applied.

The collected datasets are been taken from large real-world applications and labeled by existing detection tools, which is by far the most common way to generate code smell datasets. In supervised learning approaches, the accurate labeling of the data is an essential process to improve the accuracy of the model. Existing labeling approaches include the use of labeling tools[32,33], labeling according to custom thresholds or rules [34], and manual labeling [35]. Shen et al. [36] combined the advantages of both manual detection and heuristic detection and first used three detection tools, Anti-pattern Scanner [87], Fluid Tool, and iPlasma [2], to perform the detection, and when all detectors achieved consistent results then the results were directly marked; otherwise, they were marked as unknown. After that, experienced personnel is invited to thoroughly check the unknown code fragments according to their definition of Fowler. Finally, 644 samples with data class and 6222 samples with feature envy were obtained from each of the 8 applications, respectively. The collected datasets are relatively convenient and thus widely used at present. It is applied in 60% of the literature we collected. However, the disadvantages of this approach are the imbalance of positive and negative data samples in the datasets due to the minimum number of code smell contained in the real-world application, which

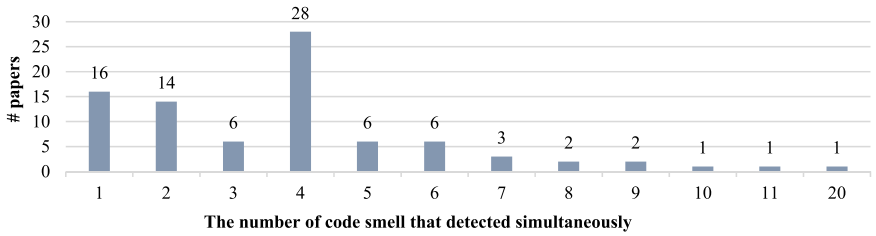

Fig. 3. The number of code smell that are detected simultaneously.

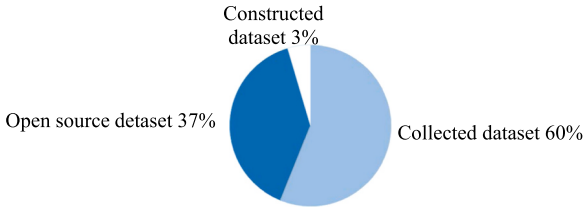

Fig. 4. The statistics of the datasets source.

will reduce the prediction accuracy of the model. Furthermore, this approach relies too much on other detection tools or manually defined thresholds, resulting in detection results that may fit the selected tools or rules.

Leveraging existing open-source datasets is also a widely used approach, with 37% of the literatures applying this approach. To find out which open source datasets are being used by researchers, we count the open source code smell datasets that have been used so far, as shown in Table 2. Fontana et al.[3] selected and labeled a total of 74 open source applications simultaneously by using a detection tool to generate a datasets containing seven types of code smell. They also classified the datasets according to the severity of code smell. This datasets is currently the most frequently used by other researchers due to its early proposal, the large variety of code smell, and the diverse sources. The datasets proposed by Palomba et al. [88] has also been applied several times. They studied 395 versions of 30 open source software systems and considered more than 10 different types of code smell. In addition, some other code smell datasets have been proposed, but they are currently limited to the authors' own work use and have not been widely used.

The constructed datasets is less applied at present. A few works use refactoring tools to generate positive samples and automatically label them to build the dataset. This approach is based on code smell generation, where code without code smell is converted into smelly by a refactoring tool for a balanced dataset. For instance, Liu et al. [38] proposed an approach to automatically generate training data without human intervention. A class m is tested by refactoring to see if it can be moved to other classes and if m is not movable, this class is skipped; if m is movable, a random test item with or without code smell is generated, and finally, a datasets is generated. The advantage of this approach is that it can automatically increase the data in the absence of positive samples of code smell, which is helpful for data balancing. However, the disadvantage of this approach is that specific refactoring tools need to be written for each code smell, and the generated code still needs to be further tested to determine whether it is smelly.

Table 2
The statistics of open source datasets been used.

The literature of open source datasets source	Frequency
Comparing and experimenting machine learning techniques for code smell detection [3]	7
On the diffuseness and the impact on maintainability of code smell: a large scale empirical investigation [88]	3
Comparing heuristic and machine learning approaches for metric-based code smell detection [64]	2
Landfill: an Open Dataset of Code Smell with Public Evaluation [89]	2
Code smell for model-view-controller architectures [90]	1
Code smell detection towards a machine learning-based approach [28]	1
Deep learning based code smell detection [37]	1
The qualitas corpus: a curated collection of java code for empirical studies [91]	1
MLCQ: Industry-relevant code smell data set [92]	1

The construction of a datasets is an essential step in supervised learning-based code smell detection. Two approaches are usually applied for data labeling: (1) direct labeling by existing detection tools; (2) manual evaluation and labeling of code fragments based on rules or directly. The former approach of automatic labeling by existing detection tools might cause the training results of the model to be influenced by the labeling tools. Due to the inconsistent outcomes of code smell detection tools, the datasets vary when different tools are employed for labeling, leading to significant variance in the training results of the model. The latter approach is similarly impacted by the individual opinions of several researchers, and the labeling outcomes can be different. Although the automatic datasets generation by refactoring approach can solve the problems of the subjectivity of code smell annotation and unbalanced datasets samples, it cannot simulate the programming habits of programmers well. In addition, the approach cannot comprehensively cover various realistic situations of code smell, making the datasets mostly similar types despite a large amount of data, which cannot meet the demand for recall in practical applications. Overall, there are advantages and disadvantages of various approaches in building datasets, and there is still much room for improvement and research, which can take the advantage of each approach to improve the quality of the datasets and make the detection results more accurate.

6. Data pre-processing

RQ3, RQ4 and RQ5 are relavant to data pre-processing techniques. RQ3 is answered in Section 6.1 while RQ4 and RQ5 are answered in Sections 6.2 and 6.3, respectively.

6.1. *Data pre-processing for different types of data*

A series of data pre-processing is often required before model training to obtain high-quality dataset. Different types of data will employ different data pre-processing strategies. Yang et al.[93] summarized the pre-processing techniques for four types of data (i.e., code, text, metrics, and graph data). We applied a similar classification to summarize the data pre-processing approaches and representative works in the selected literature. Since there is no relevant work applying graph-based datasets in our collected literature, we only analyze two types of data, textual and metrics.

The pre-processing techniques for different datasets are shown in Table 3. The pre-processing of data can be divided into four steps: data cleaning, data integration, data transformation, and data regularization. For structural-based datasets, pre-processing includes missing and abnormal values handling, duplicate and unqualified sample removal, and data regularization steps. Missing and abnormal values are generated mainly due to situations where information is temporarily unavailable or some attributes of some objects are not available. Normally,

Table 3
Data pre-processing for different datasets.

	Semantic-based datasets		Structural-based datasets	
	Specific approaches	Representative works	Specific approaches	Representative works
Data cleaning	Missing values and abnormal values handling	Jain et al.2021 [30]	Text slicing	Guo et al.2019 [31]
Data integration	Data merging	Wang et al.2019 [33]	Data merging	Guo et al.2019 [31]
	Removing duplicate data	Zhang et al.2020 [81]	Removing symbols and stop words	Zhang et al.2022 [39]
Data transformation	Normalized handling	Aleksandar et al.2022 [66]	Text vectorization	Guo et al.2019 [31]
		Gupta et al.2019 [65]		Zhang et al.2022 [39]
Data regularization	Unqualified data removal	Fontana et al. 2017 [27]	Unqualified data removal	Zhang et al.2022 [39]

these values are handled by deleting or data completion. The main reason for data normalization is the stability of the data values. After that, the optimal solution finding process will obviously become smoother and easier to converge to the optimal solution correctly. Gupta et al. [65] used the min-max approach to normalize all selected features in the same range from 0 to 1 for the collected structural-based dataset; Aleksandar et al. [66] performed the following pre-processing techniques: encoded the categorical variables using label encoding; used -1 to denote that the value of a particular metrics cannot be calculated for the analyzed code sample; normalized the metrics values using z-normalization.

There are some differences between semantic-based datasets and structural-based datasets in terms of pre-processing approaches. Semantic-based datasets are constructed by extracting valid information from the code and compressing it into the form of word vectors to be deposited in the dataset, where some steps are similar to the code-based dataset, such as data source merging, unqualified text, and duplicate instances removal. Unlike structural-based datasets, semantic-based data requires slicing the code text into sentences or words to extract the required information and performing text vectorization after removing symbols and stop words in the text [94]. The data pre-processing in literature [39] includes text splitting, removal of special and useless characters, tokenization of word sequences, and conversion of token sequences into word vectors through the pre-trained model. The literature [31] utilizes the names of items, packages, classes, and methods in the source code as text information, which is divided into a sequence of words by the word2vec algorithm, and finally converted into a word vector by using the LSTM (Long Short-Term Memory) to extract contextual relationships from the input word sequence to obtain the semantic features of the text.

6.2. Feature selection

The independent variable in machine learning and deep learning model is also called features. The selection of features plays a crucial way in the prediction of the model. The feature selected in the existing literature for different code smell detection also remains different. In order to investigate which code smell is affected by which features, we counted the correspondence between different code smell and their selected features, as shown in Table 4. As can be seen from the table, the features selected for different code smell are different. The size of a method or class, usually measured by the number of lines of code, is the most frequently selected feature metrics, and code smell such as God Class and Long Method often use these metrics. Code cohesion and coupling are also commonly considered metrics, which are often used in code smell detection such as Feature Envy and Message Chains. In addition, cyclomatic complexity and the number of domain attributes are also important metrics for assessing code smell. In addition to obtaining metrics information about the code structure, semantic related metrics features are usually obtained by analyzing the source code through natural language processing methods.

In terms of feature selection, most approaches rely only on structural information (code metrics) extracted from source code as the input of the model, for which relying solely on code metrics as features for supervised learning models is often insufficient. Code metrics information is only representative of structural features, and information regarding the deeper semantic features of the code is prone to be ignored. Only 3.8% of the related work in our collected literature extracted semantic features for model training. Palomba et al. [95] proposed TACO, an approach able to detect code smell. It does so by analyzing the properly decomposed textual blocks composing a code component, in order to apply IR methods and measure the probability that a component is affected by a given smell. The author demonstrated the usefulness of textual analysis for smell detection instantiating TACO for five code smell, i.e., *Long Method*, *Feature Envy*, *Blob*, *Promiscuous Package*, and *Misplaced Class*. Guo et al. [31] proposed a method-representation based model to represent the methods in textual code, which can effectively reflect the semantic relationships embedded in textual code, and can also automatically build a complex mapping between these features and predictions. Zhang et al. [39] propose an approach DeepSmell based on a pre-trained model and multi-level metics. They first use the static analysis tool to extract code smell instances and multi-level code metrics information in the source program and mark these instances. Secondly, the level information that relates to code smell in the source code is

Table 4
Correspondence between code smell and selected features.

Code smell	Coupling	Cohesiveness	LOC	ATFD	NOM	CC	Semantic relevance	NOP	NOA	DIT	NOC
Feature Envy	●					●	●				
Long Method			●				●				
God Class			●	●		●					
Data Class	●										
Long Parameter List								●			
Duplicated Code							●				
Large Class		●	●								
Shotgun Surgery	●										
Divergent Change							●				
Lazy Class			●						●		
Inappropriate Intimacy	●	●									
Message Chains	●	●	●								
Refused Parent Bequest					●					●	
Parallel Inheritance Hierarchies										●	●
Class Data Should Be Private									●		

parsed and obtained through the abstract syntax tree. The textual information composed of the level information is combined with code metrics information to generate the data set. Finally, text information converts into word vectors using the BERT pre-training model. To apply the GRU(Gate Recurrent Unit)-LSTM model to obtain the potential semantic relationship among the identifiers, and combine the CNN(Convolutional Neural Network) model and attention mechanism to code smell detection.

Feature selection reduces the dimensionality of features which not only makes the model easy to understand and interpret but also reduces the training time of the model. In our selected literature on supervised learning model-based code smell detection, only 25.6% of the methods (20 papers) applied feature selection. For example, Agnihotri et al. [40] used random forest and information gain feature selection approaches to extract features related to code smell detection and used random search algorithms to adjust the parameters of random forest and decision trees to achieve better performance. Jainet al. [30] implemented 32 machine learning algorithms after performing feature selection through six variations of the filter method. The author used multiple correlation methodologies to discard similar features. Mutual information, fisher score, and univariate ROC–AUC feature selection techniques were used with brute force and random forest correlation strategies. The experiments show that the performance of the model with feature selection improves in terms of Precision, F-measure, ROC-AUC values, and training time compared to the experiments without feature selection.

Feature selection techniques mainly include filter, wrapper, and embedded. Filtering is a heuristic approach, and the basic idea is to develop a criterion to measure the importance of each feature to the target attribute, and thus rank all features. Commonly used techniques include variance threshold, relevance filtering, etc. The difference between the filtering approach and the other two approaches is that its feature selection process is independent of the subsequent classifier. This approach is relatively straightforward to understand, but the effect on feature optimization and improving generalization ability cannot be guaranteed.

The basic idea of the wrapper is to take the performance of the model itself as the evaluation criterion after confirming the training model. Wrapper generally selects or excludes some features according to the objective function. The common approach is recursive feature elimination, which uses machine learning for training to obtain the weight value factors of each feature and select the feature based on it in descending order. Embedding also uses machine learning methods to select features, and each iteration uses the full set of features. This approach integrates the feature selection process with the model training process, and the feature selection is done automatically during the training process.

For the 20 papers that applied feature selection techniques, we counted the percentage of different techniques applied, as shown in Fig. 5. As seen from the figure, filtering is currently the most widely used feature selection approach, which is applied with a percentage of 63%. In contrast, the total percentage of the other two approaches is only 37%, of which wrapper accounts for 21% and embedding accounts for only 16%. In addition to the above three feature selection techniques, Hadj-Kacem et al.[41] proposed a hybrid learning-based approach that first applied autoencoder to unlabeled data to reduce its dimensionality and extract new feature representations, and then used artificial neural networks for supervised learning classification.

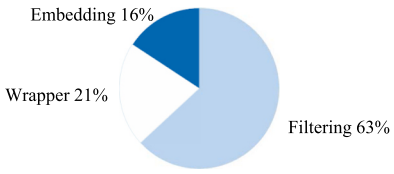

Fig. 5. Statistics on feature selection techniques.

6.3. Data balancing

Maintaining a balance between the number of positive samples (with code smell) and negative samples (without code smell) in the datasets plays a significant role in improving the accuracy of model training. Ensuring data balance while maintaining the quality of the datasets is an issue of concern for researchers. Unbalanced data may have a series of disadvantages in model training and tend to lead to overfitting of the model. The insufficient number of positive samples in the datasets will result in the features of the positive samples being incompletely learned, which will affect the final performance of the classifier (the positive samples may not be recognized). Especially for binary classification, the proportion of positive and negative samples should be relatively similar under regular situations, which is the assumption on the basis of many existing classification models.

During the construction of code smell dataset, the percentage of positive and negative samples often happens to be unbalanced, and sometimes the difference can be quite dramatic. Palombaet al. [88] pointed out that God Class accounts for less than 1% of the total classes in the real scenario, which can lead to a minimum number of positive samples in the dataset. Several researchers suggest that in real-world applications, smelly code and non-smelly code are inherently unbalanced. Although data balancing is desirable for model training, a balanced datasets does not reflect well the distribution of code smell in real-world applications, which may affect the performance of model training. The datasets in [30] were intentionally made unbalanced to depict real scenarios, and different types of smell in the same datasets were introduced by merging instances in the original class level and method level datasets. Twenty-eight papers in our selected literature applied data balancing techniques. Kaur et al. [7] applied two approaches: random undersampling is done before feature selection and feature selection is done prior to random undersampling, respectively. A total of 6 machine learning models are applied to identify change-prone classes of various versions of four open source java applications s to compare different types of machine learning algorithms on the basis of AUC, F-measure, accuracy, recall, and precision by ranking them based on their performance.

Oversampling and undersampling are widely used to ensure data balance. The undersampling technique reaches data balance by randomly sampling dropping a small number of samples from categories with a large number of samples in the dataset. However, the method drops a large amount of data randomly, which may result in an incomplete feature being learned by the model. The idea of oversampling is to add a smaller number of types of data. Instead of introducing more data to the model, this approach actually overemphasizes the positive scale data which can amplify the effect of positive scale noise on the model. Improper use of both approaches tends to cause over-fitting problems in the model.

The data synthesis technique refers to the automatic generation of similar data based on existing data to achieve data balance. SMOTE [96] is a data synthesis technique based on an improved random oversampling algorithm. Pecorelli et al. [67] investigate several approaches able to mitigate data unbalancing issues to understand their impact on machine learning based approaches for code smell detection. The results suggest that machine learning models relying on SMOTE realize the best performance. However, SMOTE randomly selects minorities as candidates to automatically generate similar samples. Not all samples are suitable to be selected for expansion. The new samples generated by SMOTE probably become closer to the boundary. As a result, the boundary between positive samples and negative samples may become disturbed, resulting in incorrect classification results.

For the 28 papers where data balancing was applied, we calculated the percentage of various data balancing techniques being applied, as shown in Fig. 6. As can be seen from the figure, 79% of the literature applied either oversampling or undersampling techniques, and the percentage of the literature using oversampling techniques is 43%

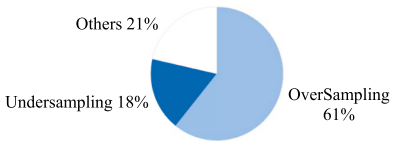

Fig. 6. Statistics of different data balancing approaches.

higher than that of the literature using undersampling techniques. Nevertheless, the majority of the current research work uses the oversampling approach, which accounts for 61%. In addition, 21% of the work uses other data balancing approaches, such as automatic data generation or assigning different weights to different samples.

7. Machine learning and deep learning models

To answer RQ6, this section presents the statistics of the existing work using machine learning models and deep learning models respectively.

7.1. Machine learning model

Machine learning-based code smell detection method is currently one of the major approaches used by researchers. We analyzed which code smell uses which machine learning model and its representative work, as shown in Table 5.

As can be seen from Table 5, existing researches mainly use machine learning models such as SVM(Support Vector Machines), Decision Trees, and Random Forests for the detection of the currently popular code smell (e.g., Long Methods, Feature Envy, and God Class). In addition for code smell such as Shotgun Surgery and Middle Man, the detection is performed using machine learning models such as K-means and Naive Bayes. Fontana et al. [3] manually labeled 1986 code smell samples in 74 software systems and applied 16 different machine learning models to detect four types of code smell: Data Class, Large Class, Feature Envy, and Long Method. All models in experiments achieved high performance on cross-validation dataset, while J48 and Random Forest had the best performance and SVM had the worst performance. Jesudoss et al. [44] used SVM and Random Forest to detect nine types of code smell. Gupta et al. [65] applied models such as Decision Tree and Logistic regression to investigate and evaluate different classification techniques, feature selection techniques, and data sampling techniques to deal with the sample imbalance in detecting seven different types of code smell. It is evident from these works that many common machine learning models (e.g., support vector machines, decision trees, and random forests) are applied to code smell detection such as Long Method, Feature Envy, and God Class, but there are still some code smell such as Inappropriate Intimacy and Parallel Inheritance Hierarchies that are not detected by applying machine learning models.

In addition to the above-mentioned works, we count the frequency with which each machine learning model is currently used by researchers further, as shown in Fig. 7. In particular, it should be noted that multiple machine learning models may be used in one research, and they will be counted separately here. As can be seen from the figure, the most frequently used model is the decision tree, which is a supervised learning model for classification problems and usually consists of a set of nodes, with splitting rules for each node for a particular feature. Amorim et al. [47] study the effectiveness of the Decision Tree algorithm to recognize code smell. For this, it was applied in a datasets containing 4 open source projects and the results were compared with the manual oracle, with existing detection approaches, and with other machine learning algorithms. The results showed that the approach was able to effectively learn rules for the detection of the code smell studied.

The random forest model is one of the more adopted machine learning models that is also used by 30 papers. It is a classifier that uses multiple decision trees to train and predict samples. Due to the use of an integrated model, random forest has a higher accuracy rate than most individual models and performs well on the test dataset. Mhawish et al. [97] proposed a machine learning and software metrics based method for code smell prediction, slightly modified the datasets of Fontana et al. [3] for the construction of two-label and multi-label datasets, and finally validated the performance of the random forest by a 10-fold cross-validation approach.

Besides the above two machine learning models with more applications, Naive Bayes is applied for 25 times in the literature. In addition, support vector machine [43], SMO(Sequential minimal optimization) [83], and KNN(K-Nearest Neighbor) [30] have been applied more than 10 times. Kaur et al. [43] proposed a support vector machine based approach to detect God class, Feature Envy, Data Class, and Long Method code smell, and validated on ArgoUML and Xerces. Other machine learning models such as AdaBoost [98] and Logistic [84] have also been applied in code smell detection.

Alazba et al.[42] proposed an integrated learning detection approach, which combines the output of a single machine learning classifier and then re-outputs it afterward to fit the predicted results to the output of all the basic classifiers. Results showed that the detection accuracy of integrated learning is higher than that of a single machine learning classifier. Fabiano et al.[64] compared the performance of traditional heuristics code smell detection and machine learning based code smell detection, which emphasized the need for further research.

7.2. Deep learning model

With the development of deep learning techniques in recent years, researchers have started using deep learning models for code smell detection. The features selected to support model training no longer simply rely on structural information, but dig deeper into the code for more information. To learn this deeper information, researchers start to use deep learning models and achieve better performance. The representative work on code smell detection based on deep learning model is presented in Table 6. Liu et al.[37] proposed a deep learning-based approach to detect code smell. Meanwhile, they proposed an approach to automatically generate smelly code and label it to construct training data, and applied it to the detection of Feature Envy, Long Methods, Large Class, and Misplaced Classes. Dong et al.[34] presented a code smell detection system with the neural network model that delivers the relationship between code smell and object-oriented metrics by taking a corpus of Java projects as an experimental dataset. Literature [81] generated a datasets by extracting more than 270,000 samples from 20 real-world applications and improved the residual network (ResNet) by introducing the metrics attention mechanism to detect Brain Class and Brain Method.

Although the above detection tools or approaches can achieve excellent performance, there are limitations in terms of detection

Table 5
Representative work on machine learning based code smell detection.

Code smell	Machine learning model	Representative work
Long Method	Support vector machines	Kaur et al., 2017 [43]
Feature Envy	Decision Tree	Alazba et al., 2021 [42]
God Class	Random Forest	Nucci D et al., 2018 [29]
Data Class	Decision Tree	Fontana et al., 2016 [3]
Long Parameter List	Support vector machines	Jesudoss et al., 2019 [44]
Large Class	Random Forest	Jesudoss et al., 2019 [44]
Duplicated Code	Support vector machine	Jesudoss et al., 2019 [44]
Shotgun Surgery	K-means	Guggulothu et al., 2019 [83]
Message Chains	Decision Tree	Wang et al., 2021 [68]
Lazy Class	Random Forest	Jesudoss et al., 2019 [44]
Refused Parent Bequest	Decision Tree	Agnihotri et al., 2020 [40]
Switch Statements	Support vector machine	Alazba et al., 2021 [42]
Divergent Change	Logistic regression	Frederico et al., 2019 [45]
Middle Man	Naive Bayes	Maneerat et al., 2011 [46]
Complex Class	Decision Tree	Gupta et al., 2019 [65]

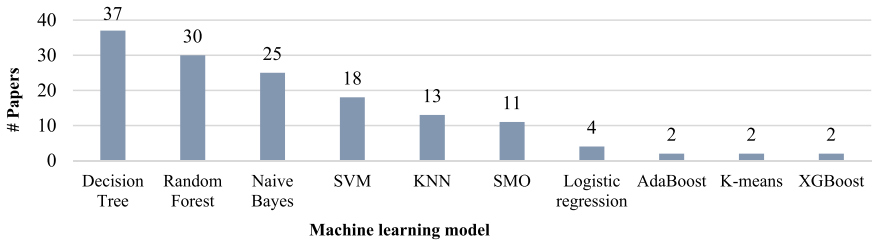

Fig. 7. Statistics on the number of papers using different machine learning models.

Table 6
Representative work on machine learning based code smell detection.

Code smell	Deep learning model	Representative work
Long Method	Convolutional neural network	Liu et al., 2019 [37]
Feature Envy	Convolutional neural network	Liu et al., 2019 [37]
God Class	Fully Connected neural network	Dong et al., 2017 [34]
Data Class	Back propagation Neural Network	Wang et al., 2019 [33]
Large Class	Back propagation Neural Network	Wang et al., 2019 [33]
Shotgun Surgery	Convolutional neural network	Lin et al., 2021 [69]
Lazy Class	Fully Connected neural network	Dong et al., 2017 [34]
Speculative Generality	Convolutional neural network	Lin et al., 2021 [69]
Complex Class	Convolutional neural network	Sharma et al., 2021 [48]
Parallel Inheritance Hierarchies	Fully Connected neural network	Dong et al., 2017 [34]
Brain Method	ResNet	Zhang et al., 2020 [81]
Brain Class	ResNet	Zhang et al., 2020 [81]

languages that most of them can only detect programs written in Java. In this scenario, Sharma et al. [48] proposed a code smell detect approach based on deep direct-learning and transfer-learning. The authors train smell detection models based on Convolution Neural Networks and Recurrent Neural Networks as their principal hidden layers along with auto-encoder models. For the first objective, they perform training and evaluation on C# samples, whereas for the second objective, they train the models from C# code and evaluate the models over Java code samples and vice-versa. The experiments show that transfer-learning is definitely feasible for implementation smell with performance comparable to that of direct-learning.

We perform statistics on various deep learning models that have been applied to code smell detection, as shown in Fig. 8. Multiple deep learning models may be applied in the same research, and separate counts are performed here. From Fig. 8, we can see that CNN, as a mature deep learning model, is the most frequently applied model. CNN has achieved great success in many research areas such as speech recognition, image segmentation, natural language processing, etc. Ananta et al. [78] used a one-dimensional convolutional neural network to detect brain class and brain method for two codes smell and obtained an accuracy of about 97% and 94%. However, the models used in their work are rather homogeneous and lack valid comparisons. Guo et al. [31] propose a method-representation based model to represent the methods in textual code, which can effectively reflect the semantic relationships embedded in textual code. They also propose a deep learning based approach that combines method-representation and a CNN model to detect feature envy. The proposed method automatically extracts semantic and structural features from the code, and can also automatically build a complex mapping between these features and predictions. A code smell detection approach proposed by Bu et al. [72] employs an automatic generation tool to generate datasets that not only utilizes ordinary software metrics but also leverages textual information in the code to mine deep semantic associations between classes using the word vectorization model word2vec. Liu et al. [37] propose an automatic approach to generating labeled training data automatically without any human intervention which creates code smell automatically by applying refactoring on well-designed source code. After that, CNN and LSTM are applied to text feature processing in the classifier, and finally, the proposed method is experimentally validated based on open source dataset.

Auto-encoder, NLP (multi-layer perception), and LSTM [99] have also been applied several times to code smell detection, the frequency is next to CNN. LSTM, which can connect information across long periods of time and thus catch long-term correlations, has shown considerable performance on a variety of datasets in representative applications. Bu et al. [72] proposed a deep learning-based God Class detection approach by applying three network models, fully connected neural networks, CNN, and LSTM, to the text feature processing part of the classifier.

ResNet, BP neural networks, and ANNs (artificial neural networks) have all been applied only once. Wang et al. [33] propose a detection method for code smell based on BP Neural Network. Four types of bad smell, Data class, God class, Long method, and Feature envy are studied and merged into method level and class level code smell datasets in their work.

7.3. Classification

The classification defines the type of the model prediction, which is mainly categorized into binary classification, multi-label classification, and severity classification. Fig. 9 gives the percentage of different classification methods used in the current main research work, where binary classification is currently the widely used classification method with 86% of the total. This classification assumes that each sample is set with a label (0 or 1) and aims to distinguish whether it is affected by code smell. This approach can only determine whether a code segment belongs to a certain type of code smell at one time.

In order to solve the problem that a segment of code in a software

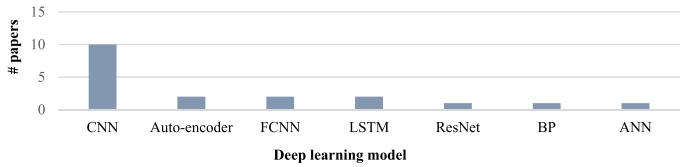

Fig. 8. Statistics on the number of papers using different deep learning models.

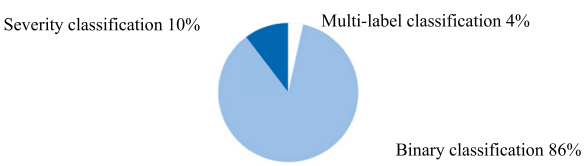

Fig. 9. Statistics of different classification approaches.

system may be affected by multiple code smell at the same time, researchers attempted to perform multi-label detection, which puts multiple code smell with high commonality in the same group, to better consider code smell correlation and improve detection efficiency. However, it is not widely used at present, and only 4% of the research works have adopted this classification approach. Wang et al. [6] proposed an ECC multi-label code smell detection method based on ranking loss. This method aims at minimizing ranking loss and chooses an optimal set of label sequences to optimize code smell detection order problem and simulate the mechanism of code smell generation by selecting random forest as the basic classifier and adopting multiple iterations of ECC to detect whether a piece of code element has long method-long parameter list, complex class-message chain or message chain-blob simultaneously. Guggulothu et al. [49] employed the conversion of a multi-label problem into a single-label problem to detect whether a given code segment is affected by multiple code smell. The authors replicated and modified the datasets by manually merging instances of other code smell datasets and experimented tree based classifier techniques on them.

A severity-based classification approach allows for an assessment of the severity of code smell and helps developers better refactor smelly code. The percentage of literature that uses this classification approach is 10%. Fontana et al. [27] described an approach for code smell detection based on machine learning-techniques, by outlining in particular how the severity of code smell can be classified through learning models. The authors approached the task as an ordinal classification (or regression) problem since severity is defined on an ordinal scale composed of four values. Liu et al. [100] presented an evolutionary version of the detection tool DT. DT can support the detection of two kinds of code smell - Duplicated code and Feature envy. In addition, the detection tool can ignore programming language, the detection method of duplicated code is used not only in JAVA but also in C++ and C.

Although the current work on supervised learning-based code smell detection has achieved significant improvements, the existing work still lacks approaches for multi-label classification according to the severity of code smell.

8. Model validation and evaluation

To answer RQ7, we have categorized the statistics of model validation strategies and evaluation approaches used in the literature we collected. Validation strategies and evaluation methods are the main ways to measure the performance of supervised learning models and are important evaluation metrics for the performance of supervised learning-based code smell detection tools.

8.1. Validation Strategy

Reliable model validation techniques ensure that the model can be fully evaluated and thus ensure the performance of the model in the practical application stage. The commonly used model validation approaches are K-fold cross-validation, random sub-sampling, leave one out, and percentage split. Random sampling validation and percentage split approach are relatively easy to operate among those approaches. The former merely selecting random samples from the datasets as the training set and the test set respectively, while the latter randomly selects the training set and the test set from the samples according to a certain percentage. These two methods have relatively high randomization, and the extracted training set is likely to have more favorable data for model training, which may lead to model overfitting problems. Compared with these two approaches, the K-fold cross-validation method can solve this problem well by setting the value of K (usually set to 5 or 10) and then using all data points in the datasets as the training set and test set respectively according to that value. Since this approach uses more datasets splitting methods for experimentation, the performance of model can be better evaluated through multiple experiments. Leave one out validation is a special case of K-fold cross-validation, which can be regarded as n-fold cross-validation when K equals n. Each data sample in K-fold cross-validation and leave-out validation is added to the training set, and the remaining data samples are selected as the test set, which ensures the reasonability of the training results to some extent. However, both those approaches make the training time longer.

We count the usage of different validation strategies in current work on supervised learning based code smell detection as shown in Fig. 10. It can be clearly seen that K-fold cross-validation is the widely utilized method at present, occupying 77% of all the literature. The other three methods have a percentage of only 23% combined, where both random sub-sampling and leave-out verification are used in only 5% of the work, and most of them are leveraged in early work. Although some of the validation strategies have been widely leveraged, some researchers have recently started to explore the shortcomings of them[23].

After a case study of 18 systems, Tantithamthavorn et al.[101] find that single-repetition holdout validation tends to produce estimates with 46–229% more bias and 53–863% more variance than the top-ranked model validation techniques. On the other hand, out-of-sample bootstrap validation yields the best balance between the bias and variance of estimates in the context of our study. Therefore, they recommend that future defect prediction studies avoid single repetition holdout validation, and instead, use out-of-sample bootstrap validation.

8.2. Evaluation strategies

There are many metrics to measure the performance of a model. Different tasks need to be measured by different metrics. It is necessary to use appropriate metrics to evaluate the model. According to different learning models, researchers may use different evaluation strategies and metrics. We count the evaluation metrics that have been applied to evaluate the performance of the model in the collected literature. In most of the literature, multiple measures are used to evaluate the model at the same time, and the counts are carried out separately here. The results are shown in Fig. 11. As can be seen from the figure, F-measure, precision, and recall are the most common model evaluation methods. Among them, precision and recall are often used at the same time, and the application frequency accounts for about 50%. Recall is defined as the proportion of the total number of correct predictions that are correctly predicted as smelly. Precision and recall affect each other. Ideally, both values are expected to be relatively high, but in practice, the two are mutually restrictive. F-measure is a good solution to the balance between recall and precision, and more than 70% of the works use F-measure to evaluate the performance of the model, which is much higher than the first two measures. In addition, precision, defined as the ratio of the number of samples correctly classified by a classifier to the total number of samples, is similarly widely used for performance evaluation, but it is used slightly less frequently than precision and recall in practical applications. The accuracy has been applied for a total of 26 times (37%).

Other model evaluation metrics emerge in an endless stream. For example, AUC-ROC is defined as the area under the ROC curve and has been used in 13 literatures. Jai et al. [30] and Gupta et al. [65] adopted three measures, accuracy, F-measure, and AUC-ROC to comprehensively evaluate the model.

Matthews Correlation Coefficient (MCC) [64,67,73] has been used in

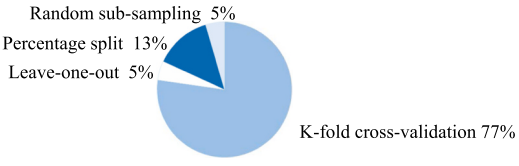

Fig. 10. Statistics of validation strategies.

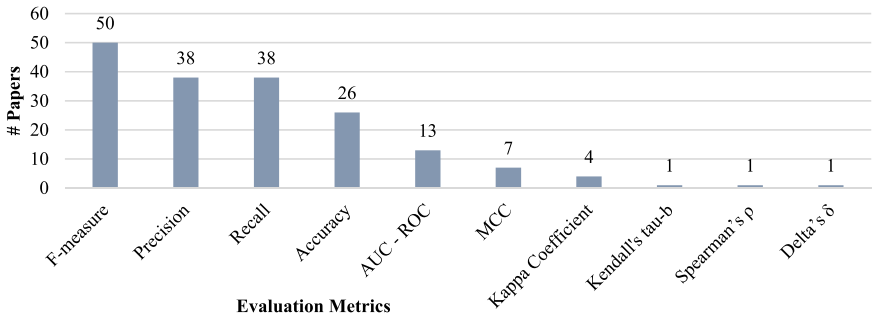

Fig. 11. Statistics of evaluation metrics.

7 papers, and it is always used together with F-measure. In terms of other measures, the Kappa coefficient [69] Kendall's tau-b [27], Spearman's ϱ, and Dirac delta function are used in 4 papers, respectively. Fontana et al. [27] employ Kendall's tau-b and Spearman's ϱ to evaluate the model. In addition, researchers also evaluated the experimental results by comparing the number of detected code smell with other tools [50].

The above-mentioned approaches evaluate the performance of supervised learning models from various aspects. However, the structural-based model evaluation method is unreasonable because it is hard to determine whether the code smell detected by the tool is really smelly, which may lead to the overestimation of the recall rate of the model.

9. Summary and prospect

Since the concept of code smell has been proposed, many researchers investigated the detection of code smell. With the development of supervised learning based code smell detection approaches, the overall number of relevant literature has shown an increasing trend. This section summarizes the related research issues and provides prospects for future work.

RQ1 presents statistics on different kinds of code smell and the number of code smell detected simultaneously in the current supervised learning based detection. The results show that there is a significant imbalance in the attention paid to different code smell in current research. Researchers pay more attention to those code smells which has a significant impact and is commonly used. Approximately 50% of the literature examines both *Feature Envy* and *Long Methods*. Code smells (such as *Middle Man* and *Inappropriate Intimacy*) receive few attention. Currently, many works do not detect only one kind of code smell but identify multiple code smell with similar characteristics. The largest number of papers detect four code smell simultaneously. Due to the significant differences in detection approaches for different code smell, the simultaneous detection of completely unrelated code smell in terms of characteristics increases the difficulty of detection. Thus, there is few works on comprehensive detection for all proposed code smell at the same time. In the future, it is necessary to take into account the balance of different code smell while enhancing the current popular smell detection and strengthening the research of detection approaches such as *Middle Man* and *Inappropriate Intimacy*. In addition, different weight values can be set according to the degree of influence of different smell on the program, so as to facilitate the subsequent optimization work of developers.

RQ2 focuses on the construction of datasets in supervised learning approaches. There are four types of datasets construction in existing works: collected datasets, open source datasets, constructed datasets, and industry datasets. Most of the datasets are extracted from open source projects and labeled. This approach usually labels the sample by manual approaches or by relying on existing detection tools. By answering this question, we analyze the advantages and disadvantages of the current approach in datasets construction and show that there is still much room for improvement and research in datasets construction. In future work, various approaches can be combined to take the strengths of each approach and weaken the effects of the weaknesses to improve the quality of the datasets and make the detection results more accurate. In addition, the existing datasets usually generates a large number of samples, and how to adopt few-shot learning to obtain better performance will be a direction of future work.

RQ3 statistics on pre-processing approaches for structural-based and semantic-based datasets. To build the dataset, it usually includes several stages such as data cleaning, integration, and transformation. By analyzing the existing work, we find that there is a lack of works related to the comprehensive pre-processing of datasets. The future work needs to enhance the pre-processing techniques for datasets to obtain sanity datasets.

RQ4 presents statistics on the features selected and the feature selection techniques applied in existing supervised learning based code smell detection approaches. The result shows that most approaches rely merely on structural information (code metrics) extracted from source code. Such approaches collect only digital features embedded in the source code. Relying solely on code metrics as features for supervised learning models is insufficient, and information about the deep semantic features of the code is prone to be ignored. In terms of feature selection techniques, the data show that the number of papers applying feature selection techniques for data dimensionality reduction is less than one-third of the total. In the future, researchers should consider aspects regarding the features of code more comprehensively and extract multimodal features to train the model for improving the accuracy of code smell detection. Meanwhile, the application of feature selection techniques should be increased to improve the model training efficiency. In addition, the metrics with a significant effect on code should be analyzed from the perspective of model interpretability in order to improve the training efficiency and stability of the model in real-world applications.

RQ5 provides a statistical analysis of the approaches to deal with data imbalance in the current work. The result indicates that about 35% of the current work deals with data imbalance. Undersampling in the existing work causes some data wastage, but the impact on the quality of the datasets is minimal compared to other approaches when a sufficient amount of data is collected for model training. In the future, large-scale data collection or automated data collection can be performed to apply undersampling for getting a high quality datasets sufficient to support model training. In addition, several studies have proposed that the balanced datasets does not reflect properly the distribution of code smell in the actual program, which may affect the performance of detection in practical applications. To address this issue, future work can investigate how to use a more realistic datasets to ensure the fitting ability of the model.

RQ6 provides statistics on the machine learning and deep learning models as well as the classification approaches. It shows that most existing works employ binary classification, while multi-label classification and severity classification have not received enough attention. The results of RQ6 indicates that the future works should be more concerned how to build model. There is a need to strengthen the researches related to the severity of code smell and the multi-label

classification of the model to help developers determine the code segments that need to be refactored efficiently and correctly. In addition, how to interpret the model while improving the performance is also a further research task in the future.

RQ7 compares model validation strategies and evaluation approaches. The widely used evaluation approaches assess the performance of supervised learning models in different aspects. However, the structural-based model evaluation approaches need further optimization. For example, code smell detection tools may have false positive cases, which can lead to an overestimation on the recall of the models. The performance of the detection tools can be analyzed more comprehensively in future work by combining automatic evaluation with manual evaluation.

10. Related works

Many researchers have conducted surveys on code smell. Min et al. [102] summarized 39 papers between 2000 and 2009, and systematically analyzed and described four aspects: the attention paid to different code smell, the aims of different studies, code smell detection methods, and the deleterious effect of code smell on software; Fernandes et al. [103] conducted a statistical analysis of 84 detection tools proposed or used in research papers were statistically analyzed in terms of code smell detection, detection methods, application languages, and so on. 29 out of them are available online for download. For 84 tools, they observe that the amount of standalone and plug-in tools is roughly the same. In addition, the review results show that Java, C, and C++ are the top three most covered programming languages for code smell detection. Most of the 84 tools are implemented in Java and rely on structural-based detection techniques. Sharma and Spinellis [104] collected relevant studies from various conferences and journals published between 1999 and 2016. The authors investigated 10 common factors that lead to the occurrence of code smell and examined existing code smell detection approaches into five categories, i.e., metrics, rules or heuristics, historical information, machine learning, and optimization-based detection. Sobrinho et al. [105] analyzed 351 research papers related to code smell published between 1990 and 2017 from five perspectives, including types of code smell, the evolution of researchers' attention, experimental settings, researchers/teams working on code smell, and paper publication distribution.

Some scholars reviewed the problems related to code smell from different aspects and sorted out the relevant research progress in recent years. In the study of Zhang et al. [21] on 8 popular Java projects with 104 released versions, an extensive empirical study is conducted to investigate 13 kinds of code smell. The authors deeply analyzed the impact of code smell on software evolution and gave relevant suggestions for developers on how to effectively refactor code in the process of software maintenance. Tian et al. [22] systematically analyze and classifies more than 300 papers related to code smell published from 1990 to June 2020. The authors analyze the development trend of code smell, quantitatively reveal the mainstream and hot spots of related research, identify the key code smell concerned by the academia, and also study the differences of concerns between industry and academia.

Some researchers have conducted systematic reviews of machine learning-based code smell detection. Caram et al. [24] categorized 26 papers from 1990 to December 2016 to statistically analyze the utilization of machine learning-based code smell detection and compare the performance of different machine learning techniques. Azeem et al. [23] studied a total of 15 pieces of literature on machine learning based code smell detection between 2000 and 2017, in terms of the examined code smell, the configuration of the machine learning model, the design of the evaluation strategy, and the performance achieved by the model were systematically analyzed.

11. Conclusion

Detecting code smell is a hot research area in software evolution. Supervised learning-based approaches have become one of the mainstream approaches for code smell detection. In this paper, we classify and analyze 86 papers of supervised learning based code smell detection, and review them from seven research questions, which are systematically analyzed and discussed in terms of datasets construction, feature selection, data balance, and model training. Finally, we summarize the problems of code smell detection based on supervised learning and suggest possible research directions. The future works include that we will analyze the correlation between different features and code smell from the perspective of model interpretability. Furthermore, we will dig into the essentials of model computation process to improve the stability of the model.

CRediT authorship contribution statement

Yang Zhang: Proposing the idea, Experimental analysis, Writing – review & editing. Chuyan GE: Conducting experimentation, Writing – original draft, review & editing. Haiyang Liu: experimentation analysis, Writing – review & editing. Kun Zheng: Writing – review & editing.

Declaration of Competing Interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Yang Zhang reports financial support was provided by Natural Science Foundation of Hebei Province, China.

Data Availability

Data will be made available on request.

Acknowledgements

The authors would like to thank the insightful comments and suggestions of those anonymous reviewers, which have improved the presentation. This work is partially supported by the Natural Science Foundation of Hebei under grant No. F2023208001 and the Overseas High-level Talent Foundation of Hebei under grant No.C20230358.

References

[1] M. Fowler, Refactoring: Improving the Design of Existing Code, Pearson Education, India, 2018.
[2] C. Marinescu, R. Marinescu, P. Mihancea, D. Ratiu, R. Wettel, Iplasma: an integrated platform for quality assessment of object-oriented design, IEEE Int. Conf. Softw. Maint. - Ind. Tool. Vol. Citeseer (2005).
[3] F. Arcelli Fontana, M.V. Mäntylä, M. Zanoni, A. Marino, Comparing and experimenting machine learning techniques for code smell detection, Empir. Softw. Eng. 21 (3) (2016) 1143–1191 [doi: 10.1007/s10664-015-9378-4].
[4] F. Palomba, R. Oliveto, A. De Lucia, Investigating code smell co-occurrences using association rule learning: a replicated study, Proc. IEEE Workshop Mach. Learn. Tech. Softw. Qual. Eval. (MaLTeSQuE). N. Y.: IEEE (2017) 8–13 [doi: 10.1109/MALTESQUE.2017.7882010].
[5] A. Garg, M. Gupta, G. Bansal, B. Mishra, V. Bajpai, Do bad smell follow some pattern?. Proceedings of the International Congress on Information and Communication Technology Springer, Singapore, 2016, pp. 39–46 [doi:10.1007/978-981-10-0767-5_5].
[6] J. Wang, J. Chen, J. Gao, Ecc multi-label code smell detection method based on ranking loss, J. Comput. Res. Dev. 58 (1) (2021) 178–188 [doi: 10.7544/issn1000-1239.2021.20190836].
[7] Kaur K., Jain S. Evaluation of machine learning approaches for change-proneness prediction using code smell. 2017.
[8] Wieman R. Anti-pattern scanner: An approach to detect anti-patterns and design violations. LAP Lambert Academic Publishing, 2011.
[9] G. Szőke, C. Nagy, L.J. Fülöp, R. Ferenc, T. Gyimóthy, Faultbuster: an automatic code smell refactoring toolset, 2015 IEEE 15th Int. Work. Conf. Source Code Anal. Manip. (SCAM) (2015) 253–258 [doi: 10.1109/SCAM.2015.7335422].
[10] T.F.M. Sirqueira, A.H.M. Brandl, E.J.P. Pedro, R. de Souza Silva, M.A.P. Araujo, Code smell analyzer: a tool to teaching support of refactoring techniques source

code, IEEE Lat. Am. Trans. 14 (2) (2016) 877–884 [doi: 10.1109/TLA.2016.7437235].
[11] A. Abuhassan, M. Alshayeb, L. Ghouti, Software smell detection techniques: a systematic literature review, J. Softw. Evol. Process (7) (2020) [doi: 10.1002/smr.2320].
[12] S.M. Olbrich, D.S. Cruzes, D.I. Sjøberg, Are all code smell harmful? A study of god classes and brain classes in the evolution of three open source systems, 2010 IEEE Int. Conf. Softw. Maint. IEEE (2010) 1–10 [doi: 10.1109/ICSM.2010.5609564].
[13] N. Moha, Y.-G. Guéhéneuc, L. Duchien, A.-F. Le Meur, Decor: a method for the specification and detection of code and design smell, IEEE Trans. Softw. Eng. 36 (1) (2009) 20–36 [doi: 10.1109/TSE.2009.50].
[14] N. Tsantalis, A. Chatzigeorgiou, Identification of move method refactoring opportunities, IEEE Trans. Softw. Eng. 35 (3) (2009) 347–367 [doi: 10.1109/TSE.2009.1].
[15] A. Ouni, M. Kessentini, H. Sahraoui, M. Boukadoum, Maintainability defects detection and correction: a multi-objective approach, Autom. Softw. Eng. 20 (1) (2013) 47–79 [doi: 10.1007/s10515-011-0098-8].
[16] S. Ma, D. Dong, Detection of large class based on latent semantic analysis, Comput. Sci. 44 (Z6) (2017) 495–498.
[17] S. Fu, B. Shen, Code bad smell detection through evolutionary data mining, 2015 ACM/IEEE Int. Symp. Empir. Softw. Eng. Meas. (ESEM). IEEE (2015) 1–9 [doi: 10.1109/ESEM.2015.7321194].
[18] S. Hong, Y. Zhang, C. Li, Y. Bai, Reinstancer: automatically refactoring for instanceof pattern matching. 2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), IEEE, Pittsburgh, PA, USA, 2022, pp. 183–187 [doi: 10.1109/ICSE-Companion55297.2022.9793818].
[19] Y. Zhang, C.S. Li, S. Shao, ReSwitcher: automatically refactoring java programs for switch expression, Proc. 32nd Int. Symp. Softw. Reliab. Eng. Workshops (ISSRE). Oct. 25-28 (2021) 399–400 (Wuhan, Hubei, China).
[20] Y. Zhang, Refactoring-based learning for fine-grained lock in concurrent programming course, Comput. Appl. Eng. Educ. 30 (2) (2022) 505–516, https://doi.org/10.1002/cae.22469.
[21] X. Zhang, C. Zhu, Empirical study of code smell impact on software evolution, J. Softw. 30 (5) (2019) 1422–1437 [doi: 10.13328/j.cnki.jos.005735].
[22] Tian Y.C., Li K.J., Wang T.M., Jiao Q.Q., Li G.J., Zhang Y.X., Liu H. A review of code smell research. Ruan Jian Xue Bao/Journal of Software, (in Chinese).
[23] M.I. Azeem, F. Palomba, L. Shi, Q. Wang, Machine learning techniques for code smell detection: a systematic literature review and meta-analysis, Inf. Softw. Technol. 108 (2019) 115–138 [doi:10.1016/j.infsof.2018.12.009].
[24] F.L. Caram, B.R.D.O. Rodrigues, A.S. Campanelli, F.S. Parreiras, Machine learning techniques for code smell detection: a systematic mapping study, Int. J. Softw. Eng. Knowl. Eng. 29 (02) (2019) 285–316 [doi:10.1142/S021819401950013X].
[25] C. Wohlin, Guidelines for snowballing in systematic literature studies and a replication in software engineering, Proc. 18th Int. Conf. Eval. Assess. Softw. Eng. N. Y. (2014) 1–10 [doi: 10.1145/2601248.2601268].
[26] S. Grant, J.R. Cordy, An interactive interface for refactoring using source transformation, 1ST Int. Workshop Refactoring: Achievements Challenges Effects (2003) 30–33 ([doi:https://doi.org/]).
[27] F.A. Fontana, M. Zanoni, Code smell severity classification using machine learning techniques, Knowl. -Based Syst. 128 (2017) 43–58 [doi:10.1016/j.knosys.2017.04.014].
[28] F.A. Fontana, M. Zanoni, A. Marino, M.V. Mäntylä, Code smell detection: towards a machine learning-based approach, IEEE Int. Conf. Softw. Maint. IEEE (2013) 396–399 [doi: 10.1109/ICSM.2013.56].
[29] D. Di Nucci, F. Palomba, D.A. Tamburri, A. Serebrenik, A. De Lucia, Detecting code smell using machine learning techniques: are we there yet? IEEE 25th Int. Conf. Softw. Anal. Evol. Reengineering (Saner.) IEEE (2018) 612–621 [doi: 10.1109/SANER.2018.8330266].
[30] S. Jain, A. Saha, Rank-based univariate feature selection methods on machine learning classifiers for code smell detection, Evolut. Intell. (2021) 1–30 [doi: 10.1007/s12065-020-00536-z].
[31] X. Guo, C. Shi, H. Jiang, Deep semantic-based feature envy identification. Proceedings of the 11th Asia-Pacific Symposium on Internetware, Association for Computing Machinery, New York, NY, USA, 2019, pp. 1–6 [doi: 10.1145/3361242.3361257].
[32] Cruz D., Santana A., Figueiredo E. Detecting bad smell with machine learning algorithms: an empirical study. In: Proceedings of the 3rd International Conference on Technical Debt. 2020. 31–40. [doi: 10.1145/3387906.3388618].
[33] S. Wang, Y. Zhang, J. Sun, Detection of bad smell in code based on bp neural network, Comput. Eng. 46 (10) (2020) 216–222 (in Chinese).
[34] D.K. Kim, Finding bad code smell with neural network models, Int. J. Electr. Comput. Eng. 7 (6) (2017) 3613–3621.
[35] M. Hozano, N. Antunes, B. Fonseca, E. Costa, Evaluating the accuracy of machine learning algorithms on detecting code smell for different developers, Int. Conf. Enterp. Inf. Syst. (2017) 474–482.
[36] L. Shen, W. Liu, X. Chen, Q. Gu, X. Liu, Improving machine learning-based code smell detection via hyper-parameter optimization. 2020 27th Asia-Pacific Software Engineering Conference (APSEC), IEEE, 2020, pp. 276–285.
[37] H. Liu, J. Jin, Z. Xu, Y. Bu, L. Zhang, Deep learning based code smell detection, IEEE Trans. Softw. Eng. 99 (2019), 1-1.
[38] Liu H., Xu Z., Zou Y. Deep learning based feature envy detection. In: Proceedings of the 33rd ACM/IEEE international conference on automated software engineering. 2018. 385–396.
[39] Zhang Y., Dong C.H., Liu H., Ge C.Y. Code smell detection approach based on pre-training model and multi-level information. J. Softw., 2022, 33(5): 1551–1568(in Chinese). ⟨http://www.jos.org.cn/1000–9825/6548.htm⟩.
[40] M. Agnihotri, A. Chug, Application of machine learning algorithms for code smell prediction using object-oriented software metrics, J. Stat. Manag. Syst. 23 (7) (2020) 1159–1171.
[41] M. Hadj-Kacem, N. Bouassida, A hybrid approach to detect code smell using deep learning, 13th Int. Conf. Eval. Nov. Approaches Softw. Eng. Madeir. Port. (2018) 137–146.
[42] A. Alazba, H. Aljamaan, Code smell detection using feature selection and stacking ensemble: an empirical investigation, Inf. Softw. Technol. 138 (2021), 106648.
[43] A. Kaur, S. Jain, S. Goel, A support vector machine based approach for code smell detection, 2017 Int. Conf. Mach. Learn. Data Sci. (MLDS). IEEE (2017) 9–14.
[44] A. Jesudoss, S. Maneesha, Identification of code smell using machine learning, 2019 Int. Conf. Intell. Comput. Control Syst. (ICCS). IEEE (2019) 54–58.
[45] F.C. Luiz, B.R. de Oliveira Rodrigues, F.S. Parreiras, Machine learning techniques for code smell detection: An empirical experiment on a highly imbalanced setup, Proc. XV Braz. Symp. . Inf. Syst. (2019) 1–8.
[46] N. Maneerat, P. Muenchaisri, Bad-smell prediction from software design model using machine learning techniques, 2011 Eighth Int. Jt. Conf. Comput. Sci. Softw. Eng. (JCSSE). IEEE (2011) 331–336.
[47] L. Amorim, E. Costa, N. Antunes, B. Fonseca, M. Ribeiro, Experience report: evaluating the effectiveness of decision trees for detecting code smell, 2015 IEEE 26th Int. Symp. Softw. Reliab. Eng. (ISSRE). IEEE (2015) 261–269.
[48] T. Sharma, V. Efstathiou, P. Louridas, D. Spinellis, Code smell detection by deep direct-learning and transfer-learning, J. Syst. Softw. 176 (2021), 110936.
[49] T. Guggulothu, S.A. Moiz, Code smell detection using multi-label classification approach, Softw. Qual. J. 28 (3) (2020) 1063–1086.
[50] K. Karađuzović-Hadžiabdić, R. Spahić, Comparison of machine learning methods for code smell detection using reduced features, 2018 3rd Int. Conf. Comput. Sci. Eng. (UBMK). IEEE (2018) 670–672.
[51] F. Pecorelli, D. Di Nucci, C. De Roover, A. De Lucia, A large empirical assessment of the role of data balancing in machine-learning-based code smell detection, J. Syst. Softw. 169 (2020), 110693.
[52] A. Kaur, S. Jain, S. Goel, Sp-j48: a novel optimization and machine-learning-based approach for solving complex problems: special application in software engineering for detecting code smell, Neural Comput. Appl. 32 (11) (2020) 7009–7027.
[53] R. Yedida, T. Menzies, How to improve deep learning for software analytics (a case study with code smell detection), arXiv Prepr. arXiv 2202 (2022) 01322.
[54] S. Jain, A. Saha, Improving performance with hybrid feature selection and ensemble machine learning techniques for code smell detection, Sci. Comput. Program. 212 (2021), 102713.
[55] H. Grodzicka, A. Ziobrowski, Z. Łakomiak, M. Kawa, L. Madeyski, Code smell prediction employing machine learning meets emerging java language constructs. Data-centric Business and Applications, Springer, 2020, pp. 137–167.
[56] Hadj-Kacem M., Bouassida N. Deep representation learning for code smell detection using variational auto-encoder. In: 2019 International Joint Conference on Neural Networks (IJCNN). IEEE, 2019. 1–8.
[57] S. Boutaib, S. Bechikh, F. Palomba, M. Elarbi, M. Makhlouf, L.B. Said, Code smell detection and identification in imbalanced environments, Expert Syst. Appl. 166 (2021), 114076.
[58] J. Nanda, J.K. Chhabra, Sshm: Smote-stacked hybrid model for improving severity classification of code smell, Int. J. Inf. Technol. (2022) 1–7.
[59] S. Dewangan, R.S. Rao, Code smell detection using classification approaches. Intelligent Systems, Springer, 2022, pp. 257–266.
[60] A. Bansal, U. Jayant, A. Jain, Categorical analysis of code smell detection using machine learning algorithms. Intelligent Sustainable Systems, Springer, 2022, pp. 703–712.
[61] S. Dewangan, R.S. Rao, A. Mishra, M. Gupta, Code smell detection using ensemble machine learning algorithms, Appl. Sci. 12 (2022) 10321, https://doi.org/10.3390/app122010321.
[62] Tomasz Lewowski Lech Madeyski, Detecting code smells using industry-relevant data, Inf. Softw. Technol. (2023) 155–107112.
[63] Amir Elmishali Bruno Sotto-Mayor, Rui Abreu Meir Kalech, Exploring design smells for smell-based defect prediction, Eng. Appl. Artif. Intell. (2022) 115–105240.
[64] F. Pecorelli, F. Palomba, D. Di Nucci, A. De Lucia, Comparing heuristic and machine learning approaches for metric-based code smell detection, Proc. 27th Int. Conf. Program Compr. (ICPC). IEEE (2019) 93–104.
[65] H. Gupta, L. Kumar, L.B.M. Neti, An empirical framework for code smell prediction using extreme learning machine, 2019 9th Annu. Inf. Technol., Electro Eng. Microelectron. Conf. (IEMECON). IEEE (2019) 189–195.
[66] A. Kovačević, J. Slivka, D. Vidaković, K.-G. Grujić, N. Luburić, S. Prokić, G. Sladić, Automatic detection of long method and god class code smell through neural source code embeddings, Expert Syst. Appl. 204 (2022), 117607.
[67] Pecorelli F., Di Nucci D., De Roover C., De Lucia A. On the role of data balancing for machine learning-based code smell detection. In: Proceedings of the 3rd ACM SIGSOFT international workshop on machine learning techniques for software quality evaluation. 2019. 19–24.
[68] Wang F., HT W., JH G. Code smell detection method based on improved C4.5 algorithm. Computer engineering and design, 2021. (in Chinese).
[69] T. Lin, X. Fu, F. Chen, L. Li, A novel approach for code smell detection based on deep leaning. EAI International Conference on Applied Cryptography in Computer and Communications, Springer, 2021, pp. 171–174.

[70] A. Patnaik, N. Padhy, Does code complexity affect the quality of real-time applications? Detection of code smell on software applications using machine learning algorithms. Proceedings of the International Conference on Data Science, Machine Learning and Artificial Intelligence, Association for Computing Machinery, New York, NY, USA, 2021, pp. 178–185.
[71] K. Alkharabsheh, Y. Crespo, M. Fernández-Delgado, J.M. Cotos, J.A. Taboada, Assessing the influence of size category of the application in god class detection, an experimental approach based on machine learning (mla), 31st Int. Conf. Softw. Eng. Knowl. Eng. (2019) 361–366.
[72] Y. Bu, H. Liu, G. Li, God class detection approach based on deep learning. Ruan jian xue bao, J. Softw. 30 (5) (2019) 1359–1374.
[73] K. Alkharabsheh, S. Alawadi, V.R. Kebande, Y. Crespo, M. Fernández-Delgado, J. A. Taboada, A comparison of machine learning algorithms on design smell detection using balanced and imbalanced dataset: a study of god class, Inf. Softw. Technol. 143 (2022), 106736.
[74] Oliveira D., Assunção W.K., Souza L., Oizumi W., Garcia A., Fonseca B. Applying machine learning to customized smell detection: A multi-application study. In: Proceedings of the 34th Brazilian Symposium on Software Engineering. 2020. 233–242.
[75] De Stefano M., Pecorelli F., Palomba F., De Lucia A. Comparing within-and cross-application machine learning algorithms for code smell detection. In: Proceedings of the 5th International Workshop on Machine Learning Techniques for Software Quality Evolution. 2021. 1–6.
[76] Y. Wang, S. Hu, L. Yin, X. Zhou, Using code evolution information to improve the quality of labels in code smell datasets, 2018 IEEE 42nd Annu. Comput. Softw. Appl. Conf. (COMPSAC). IEEE (2018) 48–53.
[77] K. Alkharabsheh, S. Almobydeen, J.A. Taboada, Y. Crespo, Influence of nominal application knowledge in the detection of design smell: an exploratory study with god class, Int. J. Adv. Stud. Comput. Sci. Eng. 5 (11) (2016) 120.
[78] A. Kaur, S. Singh, Detecting software bad smell from software design patterns using machine learning algorithms, Int. J. Appl. Eng. Res. 13 (11) (2018) 10005–10010.
[79] F. Pecorelli, S. Lujan, V. Lenarduzzi, et al., On the adequacy of static analysis warnings with respect to code smell prediction, Empir. Softw. Eng. 27 (3) (2022) 1–44.
[80] A.T. Imam, B.R. Al-Srour, A. Alhroob, The automation of the detection of large class bad smell by using genetic algorithm and deep learning, J. King Saud. Univ. - Comput. Inf. Sci. (2022).
[81] Y. Zhang, C. Dong, Mars: Detecting brain class/method code smell based on metric–attention mechanism and residual network, J. Softw.: Evol. Process (2021), e2403.
[82] Das A.K., Yadav S., Dhal S. Detecting code smell using deep learning. In: TENCON 2019–2019 IEEE Region 10 Conference (TENCON). IEEE, 2019. 2081–2086.
[83] T. Guggulothu, S.A. Moiz, Detection of shotgun surgery and message chain code smell using machine learning techniques, Int. J. Rough. Sets Date Anal. 6 (2) (2019) 34–50 [doi:10.4018/IJRSDA.2019040103].
[84] Lv T. Research on change barrier code smell detection based on logistic regression. Dalian University of Technology, 2020. (in Chinese).
[85] S. Su, Y. Zhang, D. Zhang, Coupling related code smell detection method based on deep learning, J. Comput. Appl. 42 (6) (2022) 1702–1707.
[86] B. Sotto-Mayor, A. Elmishali, M. Kalech, et al., Exploring design smell for smell-based defect prediction, Eng. Appl. Artif. Intell. 115 (2022), 105240.
[87] J.P. Reis, G.F. Carneiro, Crowdsmelling: a preliminary study on using collective knowledge in code smell detection, Empir. Softw. Eng. 27 (3) (2022) 1–35.
[88] F. Palomba, G. Bavota, M.D. Penta, F. Fasano, R. Oliveto, A.D. Lucia, On the diffuseness and the impact on maintainability of code smell: A large scale empirical investigation, Empir. Softw. Eng. 23 (3) (2018) 1188–1221.
[89] Palomba F., Di Nucci D., Tufano M., Bavota G., Oliveto R., Poshyvanyk D., De Lucia A. Landfill: an open dataset of code smell with public evaluation. In: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories. IEEE, 2015. 482–485.
[90] M. Aniche, G. Bavota, C. Treude, M.A. Gerosa, A. van Deursen, Code smell for model-view-controller architectures, Empir. Softw. Eng. 23 (4) (2018) 2121–2157.
[91] 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, 17th Asia Pac. Softw. Eng. Conf. IEEE (2010) 336–345.
[92] Madeyski L., Lewowski T. Mlcq: Industry-relevant code smell data set. In Proceedings of the evaluation and assessment in software engineering, 2020: 342–347.
[93] Yang Y., Xia X., Lo D., Grundy J. A survey on deep learning for software engineering. ACM Computing Surveys (CSUR), 2020.
[94] Y. Zhang, C. Ge, S. Hong, R. Tian, C. Dong, J. Liu, DeleSmell: code smell detection based on deep learning and latent semantic analysis, Knowl. -Based Syst. (2022), https://doi.org/10.1016/j.knosys.2022.109737.
[95] F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, A. Zaidman, A textual-based technique for smell detection, IEEE 24th Int. Conf. Program Compr. (ICPC). IEEE (2016) 1–10.
[96] N.V. Chawla, K.W. Bowyer, L.O. Hall, W.P. Kegelmeyer, Smote: synthetic minority oversampling technique, J. Artif. Intell. Res. 16 (1) (2002) 321–357.
[97] M.Y. Mhawish, Predicting code smell and analysis of predictions: using machine learning techniques and software metrics, J. Comput. Sci. Technol. 35 (6) (2020) 18.
[98] W.S. Cunha, G.A. Armijo, V.V. de Camargo, Inset: A tool to identify architecture smell using machine learning, Proc. 34th Braz. Symp. . Softw. Eng. (2020) 760–765.
[99] Y. Zhang, J. Yan, L. Qiao, H. Gao, A novel approach of data race detection based on cnn-bilstm hybrid neural network, Neural Comput. Appl. (2022) 1–15.
[100] Liu X., Zhang C. Dt: An upgraded detection tool to automatically detect two kinds of code smell: Duplicated code and feature envy. In: Proceedings of the International Conference on Geoinformatics and Data Analysis. 2018. 6–12.
[101] C. Tantithamthavorn, S. McIntosh, A.E. Hassan, K. Matsumoto, An empirical comparison of model validation techniques for defect prediction models, IEEE Trans. Softw. Eng. 43 (1) (2016) 1–18.
[102] M. Zhang, T. Hall, N. Baddoo, Code bad smell: a review of current knowledge, J. Softw. Maint. Evol.: Res. Pract. 23 (3) (2011) 179–202.
[103] Fernandes E., Oliveira J., Vale G., Paiva T., Figueiredo E. A review-based comparative study of bad smell detection tools. In: Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering. 2016. 1–12.
[104] T. Sharma, D. Spinellis, A survey on software smell, J. Syst. Softw. 138 (2018) 158–173.
[105] E.V. de Paulo Sobrinho, A. De Lucia, M. de Almeida Maia, A systematic literature review on bad smell–5 w's: which, when, what, who, where, IEEE Trans. Softw. Eng. 47 (1) (2018) 17–66.

Yang Zhang received his Ph.D. from the School of Computer, Beijing Institute of Technology. He is currently a professor at Hebei University of Science and Technology. He was a visiting scholar at Purdue University and Federation University Australia. His research interests include software refactoring and intelligent software.

Chuyan Ge received her master's degree in Hebei University of Science and Technology. She is currently a PhD candidate at Harbin Institute of Technology. Her research interests include requirements engineering and intelligent software.

Haiyang Liu is currently pursuing his master degree in the School of Information Science and Engineering, Hebei University of Science and Technology. His research interests include software refactoring and intelligent software.

Kun Zheng is currently an associate professor at Hebei University of Science and Technology. Her research interests include intelligent software.