Empirical Software Engineering (2023) 28:77
https://doi.org/10.1007/s10664-023-10312-z

Deep learning approaches for bad smell detection: a systematic literature review

Amal Alazba[1,2] · **Hamoud Aljamaan**[1] · **Mohammad Alshayeb**[1,3]

Accepted: 1 March 2023

Abstract

Context Bad smells negatively impact software quality metrics such as understandability, reusability, and maintainability. Reduced costs and enhanced software quality can be achieved through accurate bad smell detection.

Objective This review aims to summarize and synthesize the studies that used deep learning (DL) techniques for bad smell detection. Given the rapid growth of DL techniques, we believe that reviewing and analyzing the current body of knowledge would facilitate the development of new techniques and help researchers identify research gaps in this area.

Method We followed a systematic approach to identify 67 studies on DL-based bad smell detection published until October 2021. We collected and analyzed quantitative and qualitative data to obtain our results.

Results Code Clone was the most recurring smell. Supervised learning is the most adopted learning approach for DL-based bad smell detection. Convolutional neural network (CNN), Artificial neural network (ANN), Deep neural network (DNN), Long short-term memory (LSTM), Attention model, and recursive autoencoder (RAE) are the most popularly used DL models. DL models that efficiently detect bad smells, such as Tree-based CNN (TBCNN) and the Abstract syntax tree-based LSTM (AST-LSTM), tend to be specifically designed to encode features for bad smell detection.

Conclusion Many factors can affect the detection performance of DL models. Although studies exist on DL-based bad smell detection, more works that use other DL models than those already studied are needed. In this SLR, we provide a summary of existing research and recommendations for further research directions on DL-based bad smell detection.

Keywords Deep Learning · Bad Smell Detection · Systematic Literature Review

Communicated by: Denys Poshyvanyk

✉ Amal Alazba
g201901590@kfupm.edu.sa; aalazba@ksu.edu.sa

Hamoud Aljamaan
hjamaan@kfupm.edu.sa

Mohammad Alshayeb
alshayeb@kfupm.edu.sa

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

[2] Department of Information Systems, King Saud University, Riyadh 11362, Saudi Arabia

[3] Interdisciplinary Research Center for Intelligent Secure Systems, Dhahran 31261, Saudi Arabia

Published online: 11 May 2023

1 Introduction

The term "bad smell" is generally used to refer to the presence of problems related to internal design (Brown et al. 1998) and code (Fowler et al. 1999). Bad smells negatively impact software quality attributes (Pérez 2013) (Yamashita and Counsell 2013) such as understandability, reusability, and maintainability. A bad smell is not considered a fault or a problem by itself. Rather, it is an indication of a problem stemming from bad practices that may affect software maintenance and evolution (Yamashita and Counsell 2013). Automatic detection of bad smells will reduce the cost of the manual detection of bad smells in terms of time and resources. Deep learning (DL), which is an emerging area of Machine Learning (ML) that utilizes neural networks, has recently been extensively adopted for research on bad smell detection.

The present SLR focuses on DL detection approaches for design and code smells. Given the rapid growth of DL techniques, we believe that reviewing and analyzing the current body of knowledge would facilitate the development of new techniques and help researchers identify research gaps in this area. DL models are complex compared to traditional ML models; hence, building them requires the configuration of various training hyperparameters that can affect model performance. Previous reviews considered general detection approaches or focused on traditional ML detection approaches. Therefore, a study that concentrates on DL-based bad smell detection approaches and presents an in-depth analysis of the current state of the DL model performance is necessary.

The main objective of this SLR is to identify and analyze the DL models utilized in bad smell detection currently. We analyze DL model performance in relation to other aspects, such as the model's purpose, bad smells detected, datasets used for training, feature(s) used in the model, preprocessing technique(s), and encoding technique(s) used for feature transformation. We also consider all design and code smells. Our findings will allow researchers to design and implement DL models based on the most up-to-date information and practices collated from various past investigations. The results will also assist practitioners in making informed selections about the best DL models for their specific context. The following contributions are made by the present SLR to this field.

1. A set of 67 studies that address DL-based bad smell detection and were published up to October 2021 are identified; these studies can be used as the foundation for future DL-based bad smell detection research.
2. Contextual details of the DL models used in the evaluated studies are summarized: this can enable other researchers to select, design, implement, and evaluate DL models appropriate for their context.
3. The detection performance of the current state-of-the-art DL models in bad smell detection is evaluated: this evaluation is based on obtaining and compiling quantitative data (i.e., DL model performance reported in these studies in terms of the precision, recall, and f-score performance metrics).
4. Opportunities for future work that would benefit researchers and professionals who plan to work in this research area are identified.

Many studies have used DL for bad smell detection since 2016. However, only a few studies compared the study methodologies available in the literature, evaluated the performance of these methods, and discussed possible limitations and future research directions. Al-Shaaby et al. (2020) presented a systematic literature review (SLR) on code smell detection techniques using ML studies published between 2005 and 2018. However, the

main objective of their SLR is different from ours because they focused on traditional ML and did not include papers that used DL. None of the papers included in their SLR were considered in this work. Moreover, their SLR considered papers published only until 2018. More recently, Lei et al. (2022) reviewed DL application for Code Clone detection. They included 21 studies published until 2020. In comparison, various types of code smells are considered in the present review, and performance analysis is performed from various perspectives; thus, 67 papers published until October 2021 are included herein.

The rest of this paper is organized as follows: Section 2 reviews the existing SLRs and surveys; Section 3 describes the methodology used to conduct this SLR; Section 4 presents the results of the research questions; Section 5 discusses the detection performance analysis; Section 6 discusses the implications; Section 7 presents threats to validity; and Section 8 concludes this work.

2 Related Work

This section presents an overview of the previous surveys and SLRs that investigated bad smell detection and the reviews that focused on the use of ML in detecting bad smells. Various smell detection methods have been proposed (AbuHassan et al. 2021) thus far. Many of these methods suggest detecting code smells using a series of thresholds dependent on structural parameters (i.e., software metrics) to classify the core symptoms describing specific code smells (Mayvan et al. 2020). Another approach detected code smells by combining multiple structural parameters and formulating rules that vary depending on the code smell form (Moha et al. 2010). One major limitation of these methodologies is that their performance heavily depends on the threshold values and/or combination rules. Moreover, no standard threshold values and rules have been set in this regard (Lacerda et al. 2020). Accordingly, other methodologies that use ML and DL approaches have been proposed to mitigate these limitations (Al-Shaaby et al. 2020). Previous reviews investigated automatic code smell detection (Sabir et al. 2019) (Menshawy et al. 2021), some of which focused on design smell detection (Alkharabsheh et al. 2019) (Mumtaz et al. 2019).

Recent reviews investigated ML techniques for code smell detection and DL for clone detection. Sabir et al. (2019) conducted a systematic review focusing on object- and service-oriented systems by identifying and analyzing 78 papers published between 2000 and 2017 to address the evolution of code smells, detection approaches, and research trends. Menshawy et al. (2021) surveyed papers related to code smell detection that were published between 2006 and 2020 and discussed code smells, detection approaches, and automatic and semiautomatic tools for detection and refactoring. Moreover, they presented the challenges faced during code smell detection and suggested improvements to increase detection performance.

Alkharabsheh et al. (2019) reviewed the literature focused on design smell detection. They identified 395 papers between 2000 and 2017 and discussed the design smell types, detection approaches, detection tools, supported programming languages, and validation methods. An SLR proposed by Mumtaz et al. (2019) focused on design smell detection, particularly on UML model smells. Their survey covered papers published between 1990 and 2017 and reviewed the detection techniques, validation methodology, and measurement used to evaluate detection performance.

AbuHassan et al. (2021) published an SLR on the approaches used for code and design smell detection. They identified papers published up to 2018 and included 145

studies. Their review focused on the detection techniques, type of design and code smells, validation, and datasets used to evaluate detection performance.

A few SLRs and reviews investigating ML for bad smell detection have recently been proposed. Azeem et al. (2019) conducted an SLR that reviewed code smells, ML models, independent variables, datasets used, evaluation methods, and detection performance of models. Their study identified and analyzed papers published between 2000 and 2017. They found JRip and Random Forest to be the most efficient models and concluded that the use of ML for bad smell detection is limited, and its detection performance must be improved. They also found that most studies used code metrics as input for ML models.

Caram et al. (2019) reviewed code smells and explored the ML models used for bad smell detection along with their detection performance. They identified 14 papers published between 1999 and 2016 and concluded that the genetic algorithm (GA) is the most widely used technique. Their findings show that although the investigated models perform comparably, Decision Tree, K-nearest Neighbor, and Random Forest are among the best performing models.

Al-Shaaby et al. (2020) reviewed ML models used for code smell detection. They analyzed 17 papers published between 2005 and 2018 and addressed the ML models used, the most common code smells, performance measures, datasets, and ML tools. They reported that 16 various ML models were used. Of these, J48 and Random Forest performed better than the others.

A more recent review conducted by Kaur et al. (2020) discussed the use of simple and hybrid ML models for code smell detection in object-oriented systems. Their review included papers published between 2005 and 2020. They addressed the code smell, ML models, performance measures, and datasets used and found that Support Vector Machines and Decision Trees are the most common models used in the reviewed studies.

Another SLR proposed by Lewowski and Madeyski (2022) investigated artificial intelligence techniques used for code smell detection. They addressed the predictors used as input, ML/AI methods, code smell types, datasets used, and performance metrics used to evaluate the models. Their review included 45 papers published between 1998 and 2020, and their results show that most researchers used source code metrics as model input. They also found that Decision Tree and Support Vector Machines are the most widely used.

A recent review by Lei et al. (2022) investigated the use of DL for Code Clone detection. Their review covered 21 papers published between 2016 and 2020 but was limited to type 3 and 4 clones. They discussed the limitations, challenges, and recommendations for using DL in Code Clone detection.

In the present SLR, we focus on DL detection approaches and consider both design and code smell. DL models are complex compared to conventional ML models; hence, building them requires the configuration of various training hyperparameters that can affect model performance. Moreover, unlike ML models that are trained using structured data such as numerical data, DL models can be trained using complex types of data such as images, texts, and a combination of data types. Also, ML models are trained using supervised learning (i.e., a labeled dataset must be provided); however, DL models can be trained using supervised and unsupervised learning. Finally, ML models require an expert to perform feature extraction, while the process is fully automated in DL models due to their ability to learn high-level features from the data. Previous reviews considered general detection approaches or focused on conventional ML detection approaches. Our review concentrates on DL-based bad smell detection and presents an in-depth analysis of the current performance state of DL models.

3 Methodology

We followed the systematic method proposed by Kitchenham and Charters (2007) to review works on DL-based bad smell detection. This section discusses the research questions, search strategy design, selection criteria, quality assessment, and data extraction and synthesis.

3.1 Research Questions

This SLR aims to explore DL models used in bad smell detection. Table 1 presents the research questions considered herein.

3.2 Study Selection

A paper was selected if and only if it fulfilled the following inclusion criteria.

- IC1: used a DL model to detect bad smells
- IC2: reported empirical results
- IC3: a research paper that was peer-reviewed and published in a conference, journal, or workshop

The paper that fulfilled at least one of the following exclusion criteria was excluded.

- EC1: not written in English
- EC2: book chapters, work in progress, abstract, Master's thesis, or Ph.D. dissertations
- EC3: review study (e.g., SLR, survey, etc.)
- EC4: conference papers extended to a journal article of the same paper

3.3 Search Strategy Design

We identified relevant studies by selecting six digital libraries—IEEE Xplore, ACM Digital Library, Scopus, Springer Link, Science Direct, and Wiley. The search was performed in October 2021, with the inclusion of studies published up to October 2021. We also considered studies available as "early access" in the given libraries. We formulated the search string by identifying the common keywords found in the research questions that were relevant to the focus of this SLR. We focused herein on DL-based bad smell detection; thus, we identified the synonyms of bad smells, detection, and DL techniques. A pilot search performed in all specified digital libraries revised the search string to the following.

- Bad smells: ("model smell*" OR "design smell*" OR "bad smell*" OR "code smell*" OR "design flaw*" OR "antipattern*" OR "code clone*" OR "misuse case*") AND
- Detection: ("detect*" OR "predict*" OR "identif*") AND
- DL Technique: ("deep learning" OR "transfer learning" OR "representation learning" OR "CNN" OR "RNN" OR "auto-encoder*" OR "convolutional neural network*" OR "Deep neural network*" OR "Recurrent neural network*") AND Software

A total of 946 potential studies were obtained by applying the search string (Fig. 1). Note that the search string was modified according to the search restriction in each

Table 1 Targeted research questions

Research question	Rationale
RQ 1. What is the current status of deep learning-based bad smell detection?	This RQ presents the distribution of the selected primary studies (PSs) over the years
RQ 1.1. What is the distribution of the published research each year?	
RQ 2. Which bad smells have been detected using deep learning?	In this RQ, we explore the types of bad smells and the software development level at which the bad smells were detected (e.g., code level or model level). Additionally, we discuss the granularity of the detection, i.e., whether it was detected at a file, class, or method level
RQ 2.1. Which types of bad smells have been detected using deep learning-based techniques?	
RQ 2.2. What is the granularity of dependent variables that have been used in deep learning-based bad smell detection?	
RQ 3. What is the purpose of using deep learning for bad smell detection?	The main purpose of using DL techniques in the primary studies is discussed and categorized to distinguish between problems that have been resolved using DL approaches and those that remain to be resolved. In this RQ, we present the purpose of using DL and the learning approach that was followed by the PSs reviewed
RQ 4. Which input data have been used in deep learning-based bad smell detection?	In this research question, we explore the types of features that have been used as inputs for the DL models, how the data have been handled and preprocessed, and how the data were transformed into a format that is consumable by DL models
RQ 4.1. Which features have been extracted and used as input data?	
RQ 4.2. How were the data used by the PSs preprocessed?	
RQ 4.3. How are the data encoded into formats that are consumable by deep learning models?	
RQ 5. Which deep learning models are used for bad smell detection?	This RQ explores the DL models that have been used for bad smell detection in primary studies. We discuss the DL models used as predictive classification and the ones used for representation learning. Moreover, we discuss the training hyperparameters that have been considered for training the DL models
RQ 5.1. Which deep learning models are used as predictive classification models?	
RQ 5.2. Which deep learning models are used for representation learning?	
RQ 5.3. Which training hyperparameters are considered in building deep learning models?	
RQ 6. How are deep learning models evaluated and implemented in bad smell detection?	This RQ explores how the DL models were evaluated, including the validation methods used to validate the models, the evaluation metrics considered, and the baseline approaches used to compare the proposed DL models. Additionally, we discuss the availability of the source code

Table 1 (continued)

Research question	Rationale
RQ 6.1. Which validation methods are used to evaluate the deep learning models?	
RQ 6.2. Which evaluation metrics have been more frequently used to report the accuracy of deep learning models?	
RQ 6.3. Which baseline techniques are used to evaluate the deep learning models?	
RQ 6.4. Is the source code of deep learning models publicly available?	
RQ 7. How are the bad smell datasets being constructed and used?	In this RQ, we focus on the datasets used in the selected studies. The datasets used in training DL models for bad smell detection are constructed using various systems, written in different programming languages, and of different sizes. Moreover, several methods have been used to create the labels (i.e., the label), and many tools were used to extract features. The ratio of the publicly available datasets is highlighted
RQ 7.1. Which systems and datasets are used in deep learning-based bad smell detection?	
RQ 7.2. Which programming languages are most frequently used to construct datasets?	
RQ 7.3. Which approaches have been used to create the dataset?	
RQ 8. What is the current performance of bad smell detection using deep learning models?	This RQ shows the detection performance of the DL models with respect to the dataset, bad smells, DL models, encoding technique, preprocessing technique, and the features used

library, and filters were used when necessary. The exact search string used in each digital library is provided online.[1] We found approximately 2478 results on the Springer link; of these, most were irrelevant. Therefore, we filtered the results to computer science only, narrowing down the search results to 547 studies. We reduced the number of search terms used in ScienceDirect because it supports only eight (AND/OR) operators. Moreover, the asterisk (*) wildcard was limited to seven; thus, we changed the search terms accordingly.

The scanning process was conducted by the first author (a Ph.D. student). The inclusion and exclusion criteria were designed by all authors. Pilot papers were used to apply the criteria and discuss them. The search process followed for identifying relevant studies was as follows.

1) The search string was used to select relevant studies from the listed libraries. A total of 946 potential studies were obtained.
2) The inclusion/exclusion criteria were applied against the titles and the abstracts. A total of 846 studies were excluded. The duplicates were then removed (38 studies). The inclusion/exclusion criteria (IC/EC) were then applied to the whole paper. Twelve studies

[1] https://github.com/amalazba/Deep-Learning-Approaches-for-Bad-Smell-Detection-SLR

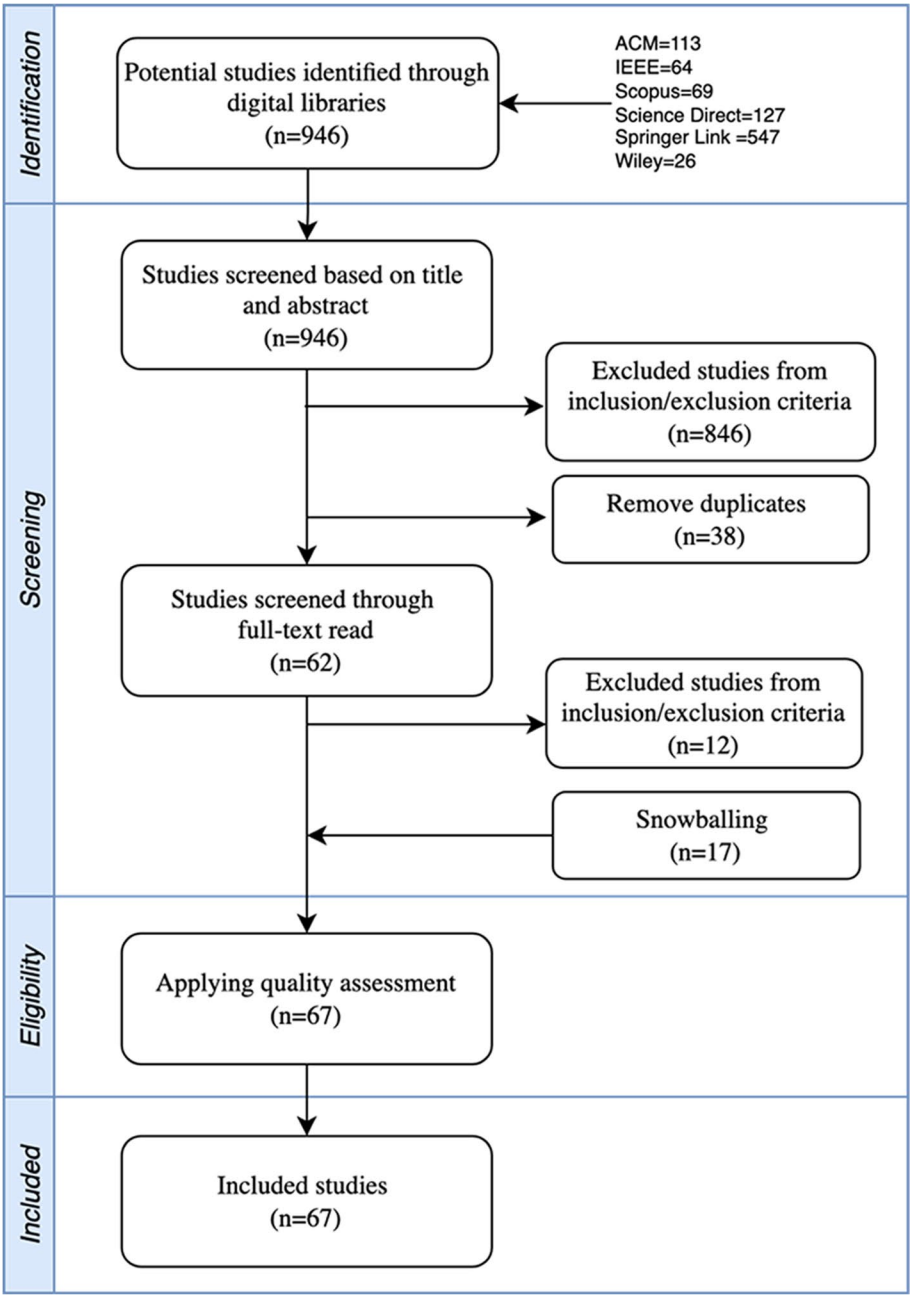


Fig. 1 Search process

Table 2 Quality assessment checklist

No.	Quality assessment checklist	Score
QA1	Are the aims and questions of the research clearly defined?	(+1) Yes/ (+0) No
QA2	Are the performance measures used to assess the models in the study specified?	(+1) Yes/ (+0) No
QA3	Are the independent variables and dependent variable(s) clearly defined?	(+1) Yes/ (+0) No
QA4	Is the validation method specified?	(+1) Yes/ (+0) No
QA5	Are the datasets adequately described?	(+1) Yes/ (+0) No
QA6	Are the details of the experiment method adequately described?	(+1) Yes/ (+0) No
QA7	Are the threats to the validity of the study specified?	(+1) Yes/ (+0) No

were excluded by applying the following filters: IC1 application excluded nine studies, IC2 application excluded one study, and EC4 application excluded two studies.

3) Next, we performed forward snowballing at one level considering all citations from the 50 relevant studies using the *Publish or Perish* tool (Anne-Wil Harzing 2016). Approximately 1000 papers were thus retrieved. Through the snowballing procedure, an additional 17 relevant studies that met our inclusion criteria were identified.
4) Finally, we ended up with 67 primary studies evaluated through quality assessment. Appendix presents the primary studies considered in this SLR following quality assessment. All papers with a quality assessment score less than the threshold were excluded from the selected studies. A total of 67 papers that passed the third round of selection were considered in the final set and used in the data extraction and synthesis.

3.4 Quality Assessment

Assessing the quality of the selected studies is essential. Quality assessment is useful in interpreting main research data and assessing the extracted data quality. The quality evaluation substantially depends on the SLR type. There are no uniform definitions of major study quality metrics (Kitchenham et al. 2009). In this review, we considered different quality criteria, including clarity of research questions, performance measures, independent/dependent variables, validation method, datasets, experiment details, and threats to validity. The quality assessment application in this work mainly aims to assist in the selection of primary studies by assessing the extracted data quality (i.e., studies with sufficient data that could answer the formulated research questions).

Table 2 shows the quality assessment criteria used to assess the studies considered in this review, wherein the maximum score was 7. We specified a threshold of 50% (i.e., any study that scored less than 3.5 on the quality assessment was excluded), as used by Wen et al. (2012). The quality assessment was performed by one author, and all of the studies were verified by the other two authors. Conflicts in the quality assessment were discussed and resolved by the authors. No paper was excluded after the assessment. Detailed quality assessment scores are provided online.[2]

[2] https://github.com/amalazba/Deep-Learning-Approaches-for-Bad-Smell-Detection-SLR

3.5 Data Extraction

We designed a data extraction form using Excel sheets to collect the data needed to answer the research questions. One of the authors was involved in the data extraction process. Accordingly, all studies were verified by the other two authors to ensure the correctness of the extracted data. Any conflicts in the extracted data were discussed and resolved. The data extraction form was structured in six sections: (1) metadata; (2) bad smells; (3) DL, (4) training considerations, (5) evaluation, and (6) datasets. Table 3 provides a detailed description of the fields of each section in the data extraction form. This form was used to extract data from the final set of the collated primary studies.

3.6 Data Synthesis

The data extracted in this SLR contained both qualitative and quantitative data (i.e., model performance). For the qualitative data, data tabulation was suitable for the collection, comparison, and synthesis of information, as prescribed in the literature (Kitchenham 2004). The data were extracted and stored in an Excel file. We then implemented an automated script using Python to compile the data and create the tables required to answer each research question.

4 Results

This section presents the results of this SLR to answer the research questions defined in Section 3.1. For each research question, the results are supported with opportunities for future work and open issues are discussed.

4.1 What is the Current Status of Deep Learning-Based Bad Smell Detection?

4.1.1 What is the Distribution of the Published Research each Year?

The selected PS years of publication ranged from 2016 to 2021. Figure 2 shows the distribution of papers published during these years. The last three years have witnessed a remarkable growth in the number of papers that used DL approaches for bad smell detection. The majority of PSs (85%) were published between 2019 and 2021; on average, approximately 19 papers per year have been published, demonstrating that the use of DL for bad smell detection is currently an active research area. White et al. [PS28] performed preliminary work in 2016. Between 2016 and 2017, DL was only applied to Code Clone smells. In 2018, other bad smells (e.g., God Class, Data Class, Feature Envy, and Long Method) were detected using DL.

DL requires a vast amount of data. Since 2016, more data on various bad smells have become publicly available. This explains why DL has been applied in this field fairly recently despite being successfully and widely used in other fields since a long time (Jaiswal et al. 2021). In 2014, Svajlenko et al. (2014) provided a benchmark open-access dataset for Code Clone, the BigCloneBench dataset. Further, Fontana et al. (2016) and Palomba et al. (2015) provided open-access datasets for various bad smells. These datasets have motivated researchers to investigate DL approaches using them.

Table 3 Data extraction form

Extracted data	Description	RQ
1. Metadata	Study ID, Title, Authors, Publication year, Publication type, Publication venue, and Summary	RQ1
2. Bad smells	Type (Code Clone, God Class…etc.), Level (code, model), and Granularity (class, method, file…etc.)	RQ2
3. Deep learning	Purpose (the goal of using DL), Features, Learning approach (supervised, unsupervised…etc.), Preprocessing (sampling, feature selection…etc.), Encoding (word2vec, code2vec…etc.), and DL models	RQ3, RQ4, RQ5.1, RQ5.2
4. Training considerations	Optimization algorithm, Loss function, Initialization, Learning rate, and Overfitting-handling	RQ5.3
5. Evaluation	Validation method, Evaluation metrics, Development environment, Source code availability, Code link, Baseline, and Model performance	RQ6, RQ8
6. Datasets	Systems/datasets, Programing language, Tools, Labeling method, Labeling tool, DS Availability, and DS link	RQ7

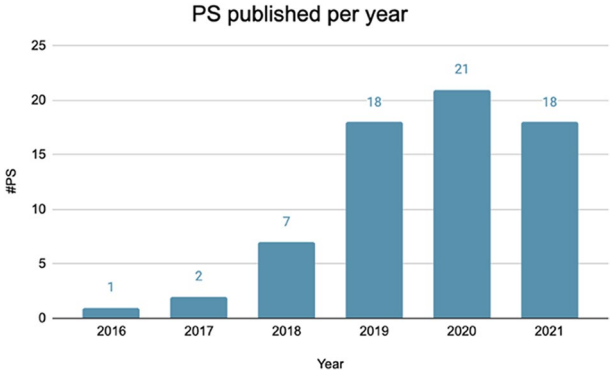


Fig. 2 Distribution of primary studies over the year

RQ1.1 summary. The use of DL for bad smell detection was introduced in 2016. Few studies were conducted between 2016 and 2018. Between 2019 and 2021, considerable research interest has been extended to the use of DL approaches for bad smell detection. The Code Clone smell was the first smell investigated using deep learning. Other smell types have been explored since 2018.

4.2 Which Bad Smells Have Been Detected Using Deep Learning?

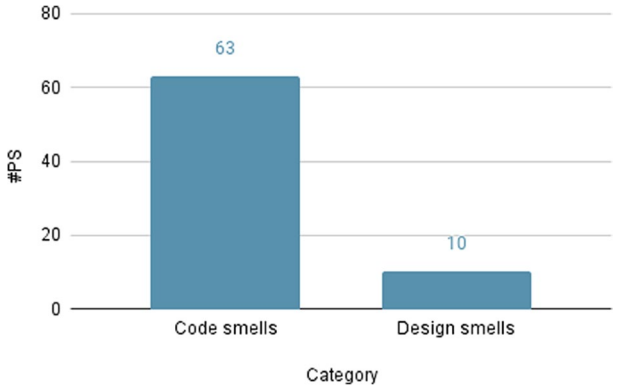
4.2.1 Which Types of Bad Smells Have Been Detected Using Deep Learning-based Techniques?

We categorized bad smells into two types based on the terms used by the authors in the PSs to describe the bad smells they are exploring. The authors in two papers referred to the smells as an "implementation smell" [PS23, PS53], whereas those in another paper called the investigated smells a "linguistic smell" [PS48]. We categorized them as code smells and design smells based on the following two definitions.

1. Code smells (Fowler et al. 1999): poor implementation choices of the source code that breach important principles, thereby negatively affecting understandability, maintainability, and reusability.
2. Design smells (Suryanarayana et al. 2015): structures in the design that suggest a violation of the primary design principles, which might negatively affect design quality.

Figure 3 shows that code smells were investigated in 63 PSs, whereas design smells were investigated in 10 PSs. Figure 4 shows the type of bad smells detected using DL (a list of PSs is presented in Appendix). The authors of the PSs used different smell names for the same problem (e.g., Blob and God Class). We decided to use the same names in this work as used in the original PSs. Among the various code smells studied, Code Clone is at the top of the list—it was studied in 58 papers out of the selected 67 PSs. The Feature Envy and Long Method code smells are next on the list, with the former being studied in nine PSs and the latter in five PSs. God Class was the most investigated design smell having been explored in four of the PSs. Blob, Deficient Encapsulation, Insufficient Modularization, and Multifaceted Abstraction design smells were investigated in two of the PSs.

Fig. 3 Bad smell categories



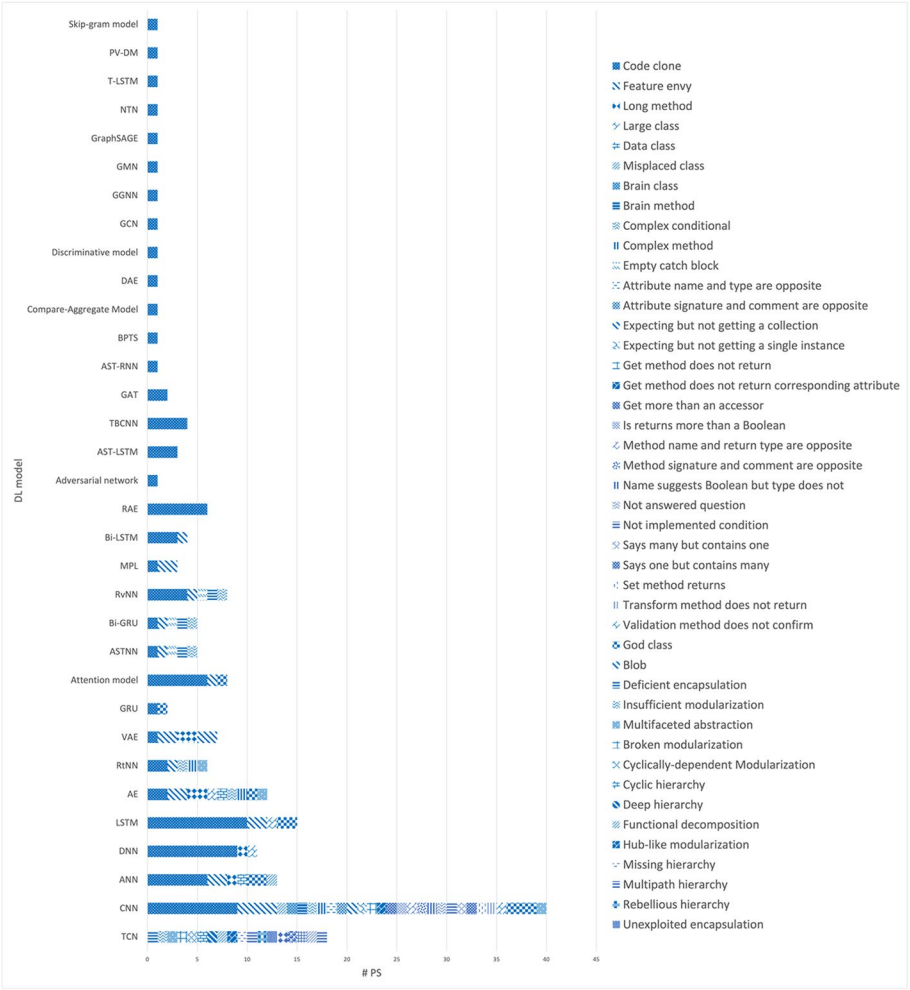


Fig. 4 Types of bad smells detected using DL

The Code Clone prevalence might be attributed to the following reasons. 1) Availability of benchmark datasets with millions of labeled instances. 2) The cost of dataset construction being less expensive than that for other smells; some studies (e.g., [PS8, PS10, PS24, PS52]) considered two code fragments that solve the same problem as clone and nonclone instances. 3) Availability of tools that can be used to find clones in the source code (Lacerda et al. 2020). 4) Detection of such bad smells across multiple programming languages (e.g., [PS11, PS18, PS24, PS35, PS45]).

The second most recurring smells (i.e., Feature Envy, Long Method, and God Class) may have attracted research attention for the following reasons.

1) Publicly available datasets (Fontana et al. 2016) (Palomba et al. 2015).
2) The existence of tools that can be used to generate labels, such as JDeodorant (Tsantalis et al. 2008), iPlasma (Marinescu et al. 2005), and DÉCOR (Moha et al. 2010). The analysis of this research question shows that further studies must focus on code smell types other than the three specified above and further investigate design smells.

Detecting bad smells in the early stages of the software development cycle has many advantages, including reduction in cost and effort. Nevertheless, bad smell detection at the model level is more difficult than at the code level (Mumtaz et al. 2019). This explains the lack of studies investigating bad smell detection at the model level. Moreover, using DL requires a large number of labeled datasets, which are not widely available (Al-Shaaby et al. 2020). This also contributes to the scarcity of such studies at the model level. Future research should thus be directed toward creating benchmark datasets and exploring DL for bad smell detection at the model level.

Many DL models have been used to detect various bad smells; however, some of them were used to detect a specific type of bad smell. For instance, CNN and LSTM have been extensively utilized to detect Code Clone, Feature Envy, and God Class smells, while autoencoders (i.e., AE and VAE) and feedforward neural networks (i.e., DNN and ANN) have been used to detect Long Method and Large Class smells. The prevalence of the CNN and LSTM might be due to their ability to extract deep semantical relationships. CNN can learn the local features of the input data, while LSTM can learn dependencies among a sequential input. All three smells (i.e., Code Clone, Feature Envy, and God Class) require semantics information for detection. Moreover, Long Method and Large Class depend on the lexical and structural information that can be detected with DL models simpler than CNN and LSTM, such as autoencoders (i.e., AE and VAE) and feedforward neural networks (i.e., DNN and ANN). A final observation is that although CNN and LSTM can extract semantics, CNN has been more popular. One possible reason for this is the training time, whereby CNN is well-suited for parallel computation, which reduces the training time.

RQ2.1 summary. We discovered that some bad smells received more research attention in the scientific literature than others. Bad smells under the code smell category are the most investigated smells. Among them, Code Clone, Feature Envy, and Long Method are the most recurring smells. The other most common smells are of the design smell category, with God Class at the top of the list. Further studies with more focus on other types of smell are needed. Only few studies have explored design smell detection. Hence, exploring DL for detecting bad smells at the model level is necessary.

Table 4 Granularity of dependent variables

Granularity	DL models*	#PS
Method	CNN (14): [PS31, PS34, PS23, PS27, PS3, PS48, PS37, PS15, PS2, PS21, PS50, PS12, PS57, PS10], DNN (9): [PS27, PS60, PS45, PS16, PS18, PS65, PS36, PS17, PS55], LSTM (9): [PS31, PS27, PS51, PS1, PS9, PS50, PS24, PS40, PS49], Attention model (7): [PS51, PS40, PS1, PS43, PS50, PS67, PS47], ANN (6): [PS4, PS41, PS40, PS66, PS44, PS24, PS32], RAE (6): [PS60, PS29, PS16, PS42, PS39, PS20], AE (4): [PS5, PS4, PS28, PS23], TBCNN (4): [PS14, PS47, PS66, PS54], AST-LSTM (3): [PS40, PS62, PS56], Bi-LSTM (3): [PS1, PS41, PS13], GCN (3): [PS67, PS40, PS42], MLP (3): [PS31, PS50, PS51], RvNN (3): [PS9, PS53, PS20], ASTNN (2): [PS9, PS53], Bi-GRU (2): [PS9, PS53], RtNN (2): [PS28, PS23], VAE (2): [PS30, PS46], Adversarial network (1): [PS56], AST-RNN (1): [PS49], BPTS (1): [PS28], Compare-Aggregate Model (1): [PS13], DAE (1): [PS57], Discriminative model (1): [PS1], GAT (1): [PS51], GGNN (1): [PS33], GMN (1): [PS33], GraphSAGE (1): [PS43], GRU (1): [PS58], NTN (1): [PS43], RvNN (1): [PS28], Skip-gram model (1): [PS63]	53
Class	CNN (6): [PS26, PS38, PS25, PS34, PS23, PS27], AE (3): [PS5, PS4, PS23], LSTM (3): [PS38, PS25, PS27], RvNN (1): [PS53], ANN (3): [PS4, PS38, PS6], ASTNN (1): [PS53], Attention model (1): [PS38], Bi-GRU (1): [PS53], DNN (1): [PS27], GGNN (1): [PS33], GMN (1): [PS33], GRU (1): [PS25], RAE (1): [PS29], RtNN (1): [PS23], TCN (1): [PS64], VAE (2): [PS30, PS46]	15
File	TBCNN (3): [PS14, PS47, PS54], LSTM (2): [PS1, PS52], AST-LSTM (2): [PS62, PS56], Attention model (2): [PS1, PS47], RvNN (2): [PS28, PS52], T-LSTM (1): [PS52], Bi-LSTM (1): [PS1], Discriminative model (1): [PS1], Adversarial network (1): [PS56], RtNN (1): [PS28], BPTS (1): [PS28], AE (1): [PS28]	8
Code fragment	ANN (1): [PS7], Bi-LSTM (1): [PS35], DNN (1): [PS7], CNN (1): [PS22], VAE (1): [PS35], GAT (1): [PS59]	4
Project	Bi-LSTM (1): [PS61], PV-DM (1): [PS11], RtNN (1): [PS19]	3
Attribute	CNN (1): [PS48]	1
Not specified	Bi-LSTM (1): [PS8]	1

*A PS may belong to one or more fields

4.2.2 What is the Granularity of Dependent Variables That Have Been used in Deep Learning-based Bad Smell Detection?

Each bad smell type can be detected on a relative scale or level of detail (i.e., granularity). Six granularities can be found in the PSs: attribute, method, class, file, project, and code fragment. An attribute is a subset of a method or a class, a method is a subset of a class, a class is a subset of a file, and a file is a subset of a project (i.e., an application). Code fragments refer to pieces of code that occur in a program. They can be a method or part of a method. They can also mean a class or part of a class. Table 4 presents the granularities with the total number of primary studies. Most of the studies (62% of the PSs) detected bad smells at the method level. Next, 18% and 9% of the PSs detected bad smells at the class and file levels, respectively. The code fragment, project, and attribute levels were the least investigated granularities in the literature, with only 5%, 4%, and 1%, respectively, of the PSs having detected bad smells at these levels. Finally, one study detected clones at multiple granularity levels, but this was not specified in the original paper.

Code Clone and Feature Envy are the only two bad smells detected at various granularity levels. Code Clone was detected at all levels except for the attribute level, whereas Feature Envy was detected at the class and method levels. An obvious reason why the remaining types of bad smells were detected at a single level is that, by definition, some bad smells can appear at one granularity level. For instance, the *Attribute name and type are opposite* is an attribute-level bad smell that cannot be detected at other granularities. Finally, we notice that no obvious relationship existed between the granularity and the selected DL models, wherein CNN was the most used at the method, class, file, and attribute levels.

RQ2.2 summary. Most studies (i.e., 62% of the PSs) detected bad smells at the method level. Next, 18% and 9% of the PSs detected bad smells at the class and file levels, respectively. The code fragment, project, and attribute levels were the least investigated granularities in the literature, with only 5%, 4%, and 1% of the PSs having detected bad smells at these levels, respectively. Future studies should investigate detection performance at various granularity levels other than the method level.

4.3 What is the Purpose of Using Deep Learning for Bad Smell Detection?

We address this research question by manually identifying the purpose of using DL in primary studies. These aims were not predetermined. They were derived from a careful reading of how each study implemented DL for bad smell detection. We identified the following three purposes.

1. **Predictive classification**: In DL, the models comprise many layers. The first few layers try to learn a useful representation of the features (implicitly) from the input. The last few layers then transform the last representation and output the predictions (Goodfellow et al. 2016). The main goal of using DL in the PSs under this category is to predict the bad smell classification. The output of the DL models is usually in the form of a binary value (i.e., 0 or 1) representing the presence or absence of a smell. The inputs are fed to the DL model, which directly outputs the classification results.
2. **Explicit representation learning**: Representation learning is described as the process of extracting useful information by learning an efficient and informative representation to improve the classification or regression model performance (Bengio et al. 2013). In this category, DL is used to transform the input into a useful representation, wherein

Table 5 Main purpose of using DL in the PSs

Purpose	PS	#PS
Explicit representation	[PS63, PS47, PS33, PS35, PS52, PS67, PS11, PS45, PS59, PS29, PS40, PS39, PS22, PS20, PS54, PS10, PS58, PS3, PS61, PS46, PS14, PS62, PS5, PS30, PS37, PS42]	26
Predictive classification	[PS18, PS23, PS34, PS65, PS55, PS15, PS64, PS7, PS49, PS27, PS2, PS17, PS25, PS26, PS44, PS6, PS8, PS12, PS19, PS48, PS21, PS36]	22
Predictive classification and explicit representation	[PS43, PS32, PS24, PS41, PS56, PS31, PS38, PS51, PS1, PS16, PS60, PS57, PS50, PS53, PS66, PS13, PS9, PS28, PS4]	19

it takes the original features as an input and outputs a feature vector. The new feature vector is then used to predict the classification using a specific measure, an algorithm, or conventional ML models that give a classification value for each bad smell. Hence, the main goal of using the DL model for bad smell detection in the PSs belonging to this category is to learn relevant input data representations rather than making predictions.

3. **Predictive classification and explicit representations**: Two different DL models can be used in this category. The first model is used to learn the representation. It outputs a feature vector, which is then fed into another DL model to make predictions. Note that more than one DL model can be used to learn the representation. The feature vectors are then concatenated and fed to the predictive model. For this category, we use the name classification and representation for conciseness in the rest of the paper.

Table 5 shows the three main purposes of using the DL models for bad smell detection. DL models are mostly used for explicit representation learning (i.e., 43% corresponds to 29 papers). Utilizing DL as a predictive classification comes next, with 33% (22 papers) of the PSs falling under this aim. Bad smell detection using DL for representation learning and predictive classification is the least explored purpose, including only 24% or 16 papers. The studies that used DL models exclusively for explicit representation used other methods to make predictions. Some PSs used similarity measures such as cosine similarity [PS10, PS14, PS22, PS33, PS35, PS37, PS42, PS45, PS47, PS52, PS54, PS58, PS59], Euclidean distances [PS39, PS63], and Manhattan distance [PS61]. Some algorithms were also considered, including the weighted L1 distance algorithm [PS3], locality sensitive hashing [PS20, PS63], near-neighbor querying algorithm [PS20], and dynamic time warping [PS42]. Finally, some PSs implemented ML classifiers, such as K-Nearest Neighbors [PS11], Random Forest [PS29], Logistic Regression [PS30, PS46, PS67], and Multilayer Perceptron (MLP) [PS31, PS40, PS50, PS51] to make predictions.

The prevalence of using DL in bad smell detection for representation learning may be attributed to the ability of selecting an appropriate DL model to represent various input formats. For example, in [PS31], they used two DL models for representation learning, LSTM for textual information and CNN for software metrics. Another example is [PS28], where RtNN and RvNN were utilized to learn useful representations from the source code and abstract syntax tree (AST), respectively. Their DL model selection was based on the fact that RtNN can model sequences of terms, such as the source code, whereas RvNN is well suited to model structures such as AST. Another possible reason behind the immense research interest in representation learning is the ability to learn a representation without supervision (i.e., the dataset does not have to be labeled), which will be discussed in the following subsection. Using DL in bad smell detection for representation learning and predictive classification has not been widely considered. An explanation for this might be the time and space complexity required to build multiple DL models (Fakhoury et al. 2018). This category lacks studies; hence, future research could focus on this purpose to show the tradeoffs between using DL models for representation and classification vs. using the DL model solely for either representation or classification.

The learning approaches of ML are grouped into many categories depending on the nature of learning. We classified the PSs based on four learning approaches: supervised, unsupervised, semi-supervised, and self-supervised. Categorization was predetermined because they are well-known and broadly used. This classification is necessary because

it will ease the identification of the nature of the problem, examine the resources, and devise a viable solution. Each learning approach is described below.

1. **Supervised** (Kotsiantis 2007): the process of learning a mapping function that maps between the input and the output. The input comprises a set of features, whereas the output comprises labels assigned to the inputs. The outputs (i.e., labels) are usually manually assigned.
2. **Unsupervised** (Gentleman and Carey 2008): the process of discovering hidden patterns and associations between the data, where it solely takes the inputs without corresponding outputs. It is called unsupervised learning because the learning process uses only a set of features without the proper labeling that guides the process.
3. **Semi-supervised** (Goodfellow et al. 2016): a learning approach that takes the benefits of supervised and unsupervised learning. It utilizes both labeled and unlabeled data to train the model, in which a small amount of labeled data with a large amount of unlabeled data during training are used.
4. **Self-supervised** (Liu et al. 2021): framing unsupervised learning problems into supervised problems and then implementing a supervised learning technique to solve them. Supervised learning techniques are used to solve a pretext task formulated from the unlabeled data. The resulting model from solving the pretext task is then used in the solution of the original (actual) modeling problem. After the training process, some labeled data are used to merely fine-tune the model.

Table 6 shows the learning approaches used by the selected PSs. Supervised learning was the most adopted learning approach in DL-based bad smell detection, representing 75% (50 papers) of the selected studies. This is followed by semi-supervised and unsupervised learning, with 13% (nine papers) and 10% (seven papers) of the total PSs having adopted these methods, respectively. Self-supervised learning was rarely used. Only 2% (one paper) of the selected studies followed this learning approach. Supervised learning was extensively used because bad smell detection is considered a binary classification problem usually tackled using this approach. However, the lack of labeled datasets urged the research community to explore alternative learning approaches. DL techniques also facilitated the adaptation of other learning approaches (Guo et al. 2020). Note that all studies utilizing the self-supervised or unsupervised learning approaches focused on detecting the Code Clone smell. In addition, 77% (seven out of nine papers) of the PSs that used a semi-supervised learning approach were also for Code Clone detection. Therefore, we recommend future studies to focus on detecting bad smells other than Code Clone and use learning approaches other than supervised learning.

One other remark is that the learning approaches are somehow related to the purpose of DL usage. Figure 5 illustrates the purpose of using DL in each learning approach. The figure clearly shows that all studies that used DL for predictive classification have followed the supervised learning approach. Conversely, in unsupervised and self-supervised learning, the DL models were mainly used for representation learning. Moreover, semi-supervised learning usually uses DL models for classification and representation. This outcome exposes a serious gap because detecting bad smells using learning approaches other than supervised learning for predictive classification should attract more research attention.

Table 7 presents the various DL models used by each learning approach. CNN and LSTM were frequently used for supervised learning, whereas ANN was mostly used in

Table 6 Learning approaches explored in the PSs

Learning approach	PS	#PS
Supervised	Predictive classification: [PS25, PS48, PS8, PS7, PS19, PS26, PS2, PS34, PS44, PS15, PS12, PS21, PS27, PS6, PS36, PS49, PS17, PS23, PS64, PS18, PS55, PS65], Classification and Representations: [PS1, PS28, PS53, PS43, PS41, PS66, PS13, PS31, PS51, PS50], Explicit representations: [PS59, PS40, PS67, PS3, PS14, PS54, PS37, PS33, PS52, PS30, PS62, PS29, PS58, PS63, PS46, PS61, PS35, PS45]	50
Semi-supervised	Classification and Representation: [PS9, PS38, PS16, PS56, PS32, PS57, PS24, PS4], Explicit representations: [PS5]	9
Unsupervised	Classification and Representation: [PS60], Explicit representation: [PS39, PS42, PS22, PS11, PS10, PS20]	7
Self-supervised	Explicit representations: [PS47]	1

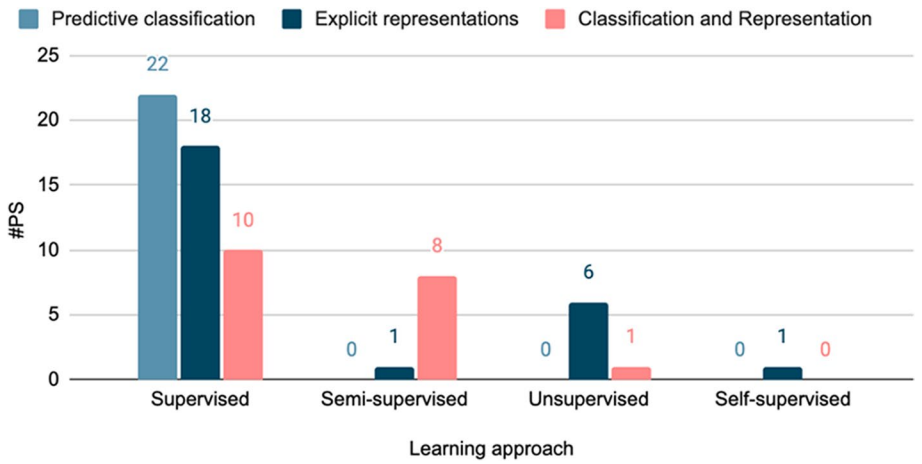


Fig. 5 Learning approach vs. purpose of DL usage

semi-supervised learning. RAE was used in unsupervised learning, while the TBCNN and the Attention model were used in self-supervised learning.

RQ3 summary. We identified three purposes of using DL for bad smell detection, one of which was more frequent than the others. The top aim is using DL for explicit representation learning. The DL model output is typically a feature vector. Utilizing DL models as predictive classifiers is the second most expressed purpose. Finally, using DL for both aims ranked last as a purpose. Future research should focus on showing the tradeoffs between using DL models for representation and classification vs.

Table 7 DL models used by learning approaches

Learning approach	DL models
Supervised	CNN (14): [PS3, PS25, PS48, PS37, PS15, PS2, PS21, PS31, PS50, PS23, PS12, PS26, PS27, PS34], LSTM (9): [PS51, PS25, PS52, PS1, PS31, PS50, PS27, PS40, PS49], DNN (8): [PS45, PS7, PS18, PS27, PS65, PS36, PS17, PS55], ANN (6): [PS41, PS7, PS66, PS44, PS40, PS6], Bi-LSTM (6): [PS41, PS61, PS1, PS35, PS13, PS8], Attention model (6): [PS1, PS43, PS50, PS67, PS40, PS51], MLP (3): [PS31, PS50, PS51], RtNN (3): [PS19, PS28, PS23], RvNN (3): [PS28, PS53, PS52], VAE (3): [PS30, PS46, PS35], TBCNN (3): [PS14, PS54, PS66], AE (2): [PS28, PS23], AST-LSTM (2): [PS40, PS62], GAT (2): [PS51, PS59], GCN (2): [PS67, PS40], GRU (2): [PS25, PS58], AST-RNN (1): [PS49], ASTNN (1): [PS53], BPTS (1): [PS28], Bi-GRU (1): [PS53], GGNN (1): [PS33], GMN (1): [PS33], GraphSAGE (1): [PS43], NTN (1): [PS43], RAE (1): [PS29], T-LSTM (1): [PS52], TCN (1): [PS64], Skip-gram model (1): [PS63], Discriminative model (1): [PS1], Compare-Aggregate Model (1): [PS13]
Semi-supervised	ANN (4): [PS4, PS38, PS32, PS24], LSTM (3): [PS9, PS38, PS24], AE (2): [PS5, PS4], CNN (2): [PS38, PS57], AST-LSTM (1): [PS56], ASTNN (1): [PS9], Bi-GRU (1): [PS9], DAE (1): [PS57], DNN (1): [PS16], RAE (1): [PS16], RvNN (1): [PS9], Attention model (1): [PS38], Adversarial network (1): [PS56]
Unsupervised	RAE (4): [PS60, PS39, PS42, PS20], CNN (2): [PS22, PS10], DNN (1): [PS60], GCN (1): [PS42], PV-DM (1): [PS11], RvNN (1): [PS20]
Self-supervised	TBCNN (1): [PS47], Attention model (1): [PS47]

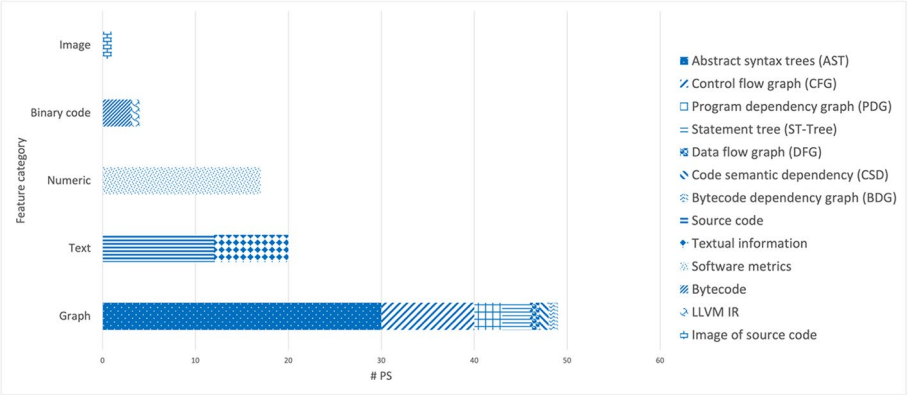


Fig. 6 Features used in DL-based bad smell detection

using them exclusively for either representation or classification. Supervised learning was the most adopted learning approach in DL-based bad smell detection, representing 75% (50 papers) of the selected PSs. Semi-supervised and unsupervised learning followed, with 13% (nine papers) and 10% (seven papers) of the considered PSs having used them, respectively. Self-supervised learning was rarely used, with only 2% (one paper) of the selected studies following this learning approach. The outcome exposes a serious gap considering that bad smell detection using learning approaches other than supervised learning for predictive classification should attract more research attention.

4.4 Which Input Data Have Been Used in Deep Learning-based Bad Smell Detection?

4.4.1 Which Features Have Been Extracted and Used as Input Data?

Various features have been utilized for bad smell detection using DL models. Figure 6 and Table 8 summarize the feature types used in the selected primary studies. The use of graph-based features was largely explored, and we found that 50% of the PSs used at least one graph type. The most frequently used graph representation is the AST. Almost one-third of the PSs (33%) have employed the AST as an input. The control flow graph (CFG) is the second most frequently used graph representation in the PSs, representing 11%.

Text- (24%) and numeric-based (20%) features rank second after the graph type. Text-based features mainly include the source code and the textual information extracted from the source code or other references. Some examples of textual information are the identifiers name [PS29], method and class names [PS31, PS41], comments [PS48], and text from the API documentation [PS11]. Numerical features include various software metrics calculated from the code [PS38, PS18, PS41, PS12, PS64, PS4, PS27, PS26, PS31, PS46, PS34, PS5, PS55, PS25, PS7, PS65] or from the model [PS6]. Binary code and images are rarely used as inputs in DL-based bad smell detection. In fact, only 4 and 1% of the PSs employed binary code and images, respectively.

We observed that the majority of papers (73% PSs) utilized only one feature, while the rest (27% PSs) used a combination of at least two features [PS7, PS8, PS12, PS15,

Table 8 Features used in DL-based bad smell detection

Type	#PS	Features	PS *	#PS
Graph	43	Abstract syntax trees (AST)	CNN (6): [PS12, PS15, PS2, PS37, PS3, PS57], TBCNN (4): [PS66, PS47, PS54, PS14], ANN (4): [PS66, PS44, PS40, PS24], RAE (4): [PS39, PS16, PS60, PS29], LSTM (3): [PS49, PS52, PS24], AST-LSTM (3): [PS62, PS56, PS40], Attention model (3): [PS47, PS40, PS67], Bi-LSTM (3): [PS8, PS35, PS13], VAE (3): [PS46, PS35, PS30], DNN (2): [PS60, PS16], RvNN (2): [PS28, PS52], GCN (1): [PS67], Adversarial network (1): [PS56], AE (1): [PS28], AST-RNN (1): [PS49], BPTS (1): [PS28], Compare-Aggregate Model (1): [PS13], DAE (1): [PS57], GGNN (1): [PS33], GMN (1): [PS33], Skip-gram model (1): [PS63], T-LSTM (1): [PS52]	30
		Control flow graph (CFG)	ANN (3): [PS40, PS32, PS44], Attention model (2): [PS43, PS40], GCN (2): [PS40, PS42], RAE (2): [PS42, PS29], DNN (1): [PS36], GAT (1): [PS59], GraphSAGE (1): [PS43], GRU (1): [PS58], NTN (1): [PS43], RtNN (1): [PS19]	10
		Program dependency graph (PDG)	CNN (2): [PS12, PS15], Attention model (1): [PS51], GAT (1): [PS51], LSTM (1): [PS51], MLP (1): [PS51]	3
		Statement tree (ST-Tree)	ASTNN (2): [PS9, PS53], Bi-GRU (2): [PS9, PS53], LSTM (2): [PS1, PS9], RvNN (2): [PS9, PS53], Attention model (1): [PS1], Bi-LSTM (1): [PS1], Discriminative model (1): [PS1]	3
		Data flow graph (DFG)	ANN (1): [PS32]	1
		Code semantic dependency (CSD)	Attention model (1): [PS50], CNN (1): [PS50], MLP (1): [PS50]	1
		Bytecode dependency graph (BDG)	CNN (1): [PS15]	1
Text	20	Source code	CNN (4): [PS23, PS22, PS10, PS25], Bi-LSTM (3): [PS8, PS35, PS61], AE (2): [PS23, PS28], ANN (2): [PS7, PS40], DNN (2): [PS7, PS17], LSTM (2): [PS40, PS25], RtNN (2): [PS23, PS28], Attention model (1): [PS40], BPTS (1): [PS28], GRU (1): [PS25], RAE (1): [PS39], VAE (1): [PS35]	12
		Textual information	LSTM (4): [PS31, PS50, PS38, PS27], CNN (3): [PS38, PS27, PS48], MLP (2): [PS31, PS50], Attention model (2): [PS38, PS50], RAE (1): [PS29], PV-DM (1): [PS11], DNN (1): [PS27], Bi-LSTM (1): [PS41], ANN (1): [PS38]	8
Numeric	17	Software metrics	CNN (7): [PS38, PS12, PS31, PS25, PS27, PS34, PS26], ANN (5): [PS41, PS38, PS4, PS6, PS7], DNN (5): [PS55, PS7, PS65, PS18, PS27], LSTM (3): [PS38, PS27, PS25], AE (2): [PS4, PS5], Attention model (1): [PS38], GRU (1): [PS25], MLP (1): [PS31], TCN (1): [PS64], VAE (1): [PS46]	17
Binary code	4	Bytecode	RAE (2): [PS20, PS29], ANN (1): [PS32], RvNN (1): [PS20]	3
		LLVM IR	DNN (1): [PS45]	1
Image	1	Image of source code	CNN (1): [PS21]	1

*A PS may belong to one or more fields

PS25, PS27, PS28, PS29, PS31, PS32, PS35, PS38, PS39, PS40, PS41, PS44, PS46, PS50]. Using multiple features is highly recommended because various features capture different information types. For instance, the AST can capture the structural features of the source code, but textual information is needed to capture the semantics information. Combining multiple features is effective in increasing the DL model performance compared to relying solely on one feature [PS25]. Therefore, we highly recommend that future studies explore the use of multiple features for bad smell detection. Future research might also examine the feature type well suited for each smell type considering that some bad smells can be detected on the structure level (e.g., Long Method), while others need semantics information (e.g., Feature Envy).

Among the utilized DL models, the CNN and the LSTM are the most frequently used on graph-based features. The LSTM was frequently used for text-based features, while the CNN was mostly used for numeric- and image-based features. Finally, the RAE was used for binary code features. The feature type must be considered when selecting DL models because each model has characteristics better suited for a specific type of data. For instance, the CNN is useful with data that have a spatial relationship like images and speech, whereas the LSTM is designed to work with sequential data, such as text. The graph neural network (GNN) is the most suitable model for graph-based features like AST, CFG, and PDG; however, only a very few studies have considered using it (GGNN: [PS33], GraphSAGE: [PS43], GMN: [PS33]) because it is still evolving. We recommend that future research to explore and develop GNN-based DL modes (e.g., gated graph sequence neural networks (Li et al. 2017a, b)) to better represent the graph-based features.

Table 9 Data preprocessing and techniques used

Data preprocessing	Techniques (#PS): *
Data balancing	Undersampling (4): [PS23, PS44, PS21, PS66], Upsampling (2): [PS42, PS52], SMOTE (2): [PS40, PS31], NA(2): [PS53, PS49]
Feature selection	Autoencoder (1): [PS4], Chi-square (1): [PS6], Gain ratio (1): [PS46], Recursive feature elimination (RFE) (1): [PS5]
Tokenization	Splitting source code to tokens (11): [PS7, PS21, PS23, PS25, PS61, PS40, PS35, PS28, PS17, PS8, PS10]
Data normalization	Special symbols removal (5): [PS19, PS28, PS48, PS66, PS67], Comment removal (4): [PS18, PS7, PS44, PS66], Trimming (2): [PS15, PS12], Min–max normalization (2): [PS15, PS64], Stop nodes removal (2): [PS29, PS63], Whitespace and newline removal (2): [PS66, PS39], Pretty-printing (2): [PS66, PS7], Stop words removal (1): [PS19], Lowercase (1): [PS25], Not specified: (1): [PS1]
Filtering	Methods' lines of code (LOC) (2): [PS17, PS40], Number of tokens (1): [PS28], Singular value decomposition (SVD) (1): [PS11], Action filter based on API call similarity (1): [PS18], Mean input size (1): [PS23], Removing data based on predefined assumptions (1): [PS27], Number of nodes in a tree (1): [PS49], Action filter based on methods similarity (1): [PS55], Size similarity filter (1): [PS55]
Data transformation	Transforming identifier names and literal values (7): [PS12, PS29, PS7, PS42, PS66, PS44, PS39], Part of speech tagging (1): [PS48], Splitting individual terms (1): [PS48]
Not mentioned	Not mentioned: (31): [PS2, PS20, PS50, PS57, PS24, PS34, PS51, PS56, PS43, PS30, PS45, PS32, PS47, PS3, PS60, PS62, PS37, PS41, PS14, PS9, PS38, PS33, PS54, PS65, PS36, PS22, PS26, PS16, PS58, PS13, PS59]

*A PS may belong to one or more fields

RQ4.1 summary. The use of graph-based features was largely explored. We found that 50% of the PSs used at least one graph type. The text- (24%) and numeric-based (20%) features ranked second after the graph-based features. Binary code and images were rarely used as inputs in DL-based bad smell detection. Only 4 and 1% of the PSs employed binary code and images, respectively. Future research might explore the feature type well suited for each smell type considering that some bad smells can be detected on the structure level (e.g., Long Method), while others need semantics information (e.g., Feature Envy).

4.4.2 How were the Data used by the PSs Preprocessed?

Many studies addressed data preprocessing steps, including data balancing, feature selection (FS), tokenization, data normalization, filtering, and data transformation, to handle data issues. Table 9 describes these data preprocessing steps along with the techniques used in the selected primary studies. A precisely stated technique name was presented along with the study. Three studies [PS53, PS49, and PS1] did not address the technique name, but mentioned to have used data balancing and normalization techniques; hence, we reported the techniques as not available (NA). Accordingly, 54% of the selected PSs used at least one data preprocessing method, while 46% did not include data preprocessing methods.

Data Balancing In most cases, bad smells are not uniformly distributed, and the number of smelly instances is considerably lower than the number of non-smelly ones. This problem is known as the class imbalance problem (Pecorelli et al. 2020), where the minority class becomes difficult to detect when trained on imbalanced data because the model is biased and in favor of the majority class. A data sampling technique is typically used to change the distribution of smelly and non-smelly instances. Few studies addressed this issue, and various techniques were used to overcome it, including undersampling [PS23, PS44, PS21, PS66], upsampling [PS42, PS52], and Synthetic minority over-sampling (SMOTE) [PS40, PS31]. In undersampling, a subset of the majority class is randomly selected and then discarded to achieve a balanced proportion. Conversely, in upsampling, a random subset of the minority class instances is duplicated and added to the actual dataset. In SMOTE, new artificial instances of the minority class are generated based on the feature similarity.

Feature Selection Irrelevant and redundant features might affect the performance of ML models (Karabulut et al. 2012). Few studies used the FS technique to solve this issue. FS is the process of finding and eliminating duplicated and irrelevant features to have a final subset of features that yield better predictive performances. Applying feature selection is a reasonable way of handling data with many features. Various feature selection techniques have been used, including autoencoders [PS4], chi-square [PS6], gain ratio [PS46], and recursive feature elimination (RFE) [PS5].

Tokenization Tokenization is the process of dividing a sequence of characters to a sequence of tokens. This preprocessing method was utilized in 11 PSs [PS7, PS21, PS23, PS25, PS61, PS40, PS35, PS28, PS17, PS8, and PS10].

Data Normalization The top two normalization techniques used are special symbol removal and comment removal. Five studies normalized the source code by removing

special symbols like '<', '>', ':' … etc. [PS19, PS28, PS48, PS66, and PS67], while four PSs excluded comments [PS18, PS7, PS44, and PS66].

Filtering Various filtering techniques were employed to filter out odd or trivial instances from the data and prevent model overfitting. Data filtering by removing instances based on the methods' lines of code (LOC) was considered in two PSs [PS17 and PS40]. Meanwhile, [PS40] excluded instances that had less than or equal to five LOC. Instance removal based on the number of tokens was performed in [PS28], which excluded instances that had less than 50 tokens.

Data Transformation Some studies transformed the source code by adding or modifying the identifier names and the literal values by replacing variables and values by their type, such as ⟨int⟩, ⟨string⟩, … etc. [PS12, PS29, PS7, PS42, PS66, PS44, PS39], tagging each token with its part of speech [PS48] or splitting individual terms into one or more terms [PS48].

One study [PS21] concluded that using the undersampling data balancing technique yielded better and more reliable results. However, we cannot derive a solid conclusion because only a few studies have explored the class imbalance issue. Comprehensive and empirical studies are needed to investigate the impact of various data balancing and feature selection techniques on the DL models for bad smell detection. Most studies did not specify whether or not they handled data-related issues, which might be due to the fact that DL models are relatively robust to noise compared to conventional ML approaches (Khan et al. 2018). Although some studies normalized the source code to minimize memory and processing consumption, the techniques used might have affected the detection performance. Further experiments are needed to show the feasibility of such techniques. For instance, the removal of special symbols is a well-known and -established normalization technique in NLP, but its application to the source code may not be suitable because unlike in natural language, special symbols have meanings in programming languages. Nevertheless, data issues can dramatically affect the ML model performance. Future studies related to DL-based bad smell detection should investigate the robustness of DL models.

RQ4.2 summary. Fifty-four percent of the PSs addressed the data preprocessing methods, while 46% did not include them. The frequently used techniques for class imbalance were the undersampling, upsampling, and SMOTE techniques. Autoencoders, chi-square, gain ratio, and recursive feature elimination (RFE) were employed to address the irrelevant and redundant feature issue. Other preprocessing methods like tokenization, data normalization, filtering, and data transformation have been used, but further empirical studies are needed to investigate the impact of these techniques on the DL models for bad smell detection.

4.4.3 How are the Data Encoded into Formats that are Consumable by Deep Learning Models?

After feature selection and processing, some of the features must be encoded to fit the neural network requirements. Encoding techniques are used to transform features like graphs and words into appropriate input (i.e., numerical representation). Table 10 presents the encoding techniques utilized in the identified PSs. Thirty-seven percent of the selected

Table 10 Encoding techniques used in the PSs

Category	Encoding	PS *	#PS
Graph-based	Ast2vec	[PS66]	1
	Graph embedding technique (HOPE)	[PS29]	1
	Graph2vec	[PS44]	1
	Tree-based skip-gram	[PS24]	1
Text-based	Code2vec	[PS23]	1
	Coding criterion	[PS30]	1
	Continuous bags of words (CBOW)	[PS61]	1
	Doc2vec	[PS11, PS22, PS40]	3
	One-hot	[PS28, PS43, PS51, PS20]	4
	Position aware character embedding (PACE)	[PS54]	1
	Sent2vec	[PS63]	1
	Skip-gram	[PS63]	1
	TF-IDF	[PS39, PS19, PS67, PS10, PS25]	5
	Word2vec	[PS9, PS53, PS60, PS13, PS31, PS58, PS38, PS18, PS48, PS67, PS49, PS44, PS40, PS2, PS37, PS39, PS16, PS41, PS57, PS27, PS50, PS62, PS56]	23
Not used	-	[PS36, PS55, PS35, PS14, PS4, PS52, PS1, PS5, PS34, PS12, PS59, PS33, PS47, PS32, PS64, PS3, PS17, PS45, PS46, PS42, PS15, PS8, PS26, PS21, PS7, PS6, PS65]	27

*A PS may belong to one or more fields

PSs did not employ any encoding techniques, while 63% considered one or more encoding strategies. Most of the studies used text-based encoding (57%). Among them, word2vec was the dominant embedding technique, representing 32% of usage. This was followed by the term frequency–inverse document frequency (TF–IDF), one-hot, and Doc2vec techniques representing 7, 6, and 4% of the studies, respectively. The graph-based embedding techniques were used in 6% of the primary studies.

Although many studies used graph-based features (RQ4.1), most of them used text-based encoding techniques for feature embedding. This shows the need for new embedding techniques dedicated to encoding the structural representation of the source code. Only one study utilized a code-specific encoding technique (i.e., code2vec [PS23]). The majority used conventional encoding techniques to encode natural language text. Embedding for natural language text usually employs a vocabulary size from thousands to millions. In programming languages, however, the vocabulary size is considered smaller. Therefore, further research for encoding the source code would be interesting.

RQ4.3 summary. Approximately 37% of the papers did not employ any encoding techniques, while 63% considered one or more. The majority of the studies used text-based encoding (57%). word2vec was the dominant embedding technique, with a percentage of 32%. Graph-based embedding techniques were used in 5% of the primary studies. New embedding techniques dedicated to source code encoding are needed.

4.5 Which Deep Learning Models are used for Bad Smell Detection?

4.5.1 Which Deep Learning Models are used as Predictive Classification Models?

Figure 7 and Table 11 summarize the studies that employed DL models for classification in bad smell detection, categorized into the main DL family that each model belongs to (Sarker 2021), along with the primary studies. The DL models that did not belong to any category were specified as "others." Feed-forward neural network-based DL models were the most widely explored category (48%), possibly due to network simplicity. It does not have additional layers like convolution or pooling layers. Convolutional network-based models represent the second most frequently used category (24%). Recurrent networks (18%) and autoencoders (8%) were the two least used categories. Among the 17 DL models shown in Table 11, the three most frequently employed DL models were CNN (11), ANN (10), and DNN (9). The remaining DL models were explored in up to three primary studies.

RQ5.1 summary. Feed-forward neural network-based models were the most widely explored category (48%). The total number of DL models used in the PSs was 17 models. The CNN (11), ANN (10), and DNN (9) were the most popular DL models used to build prediction classification models for bad smell detection. Many DL models have not been

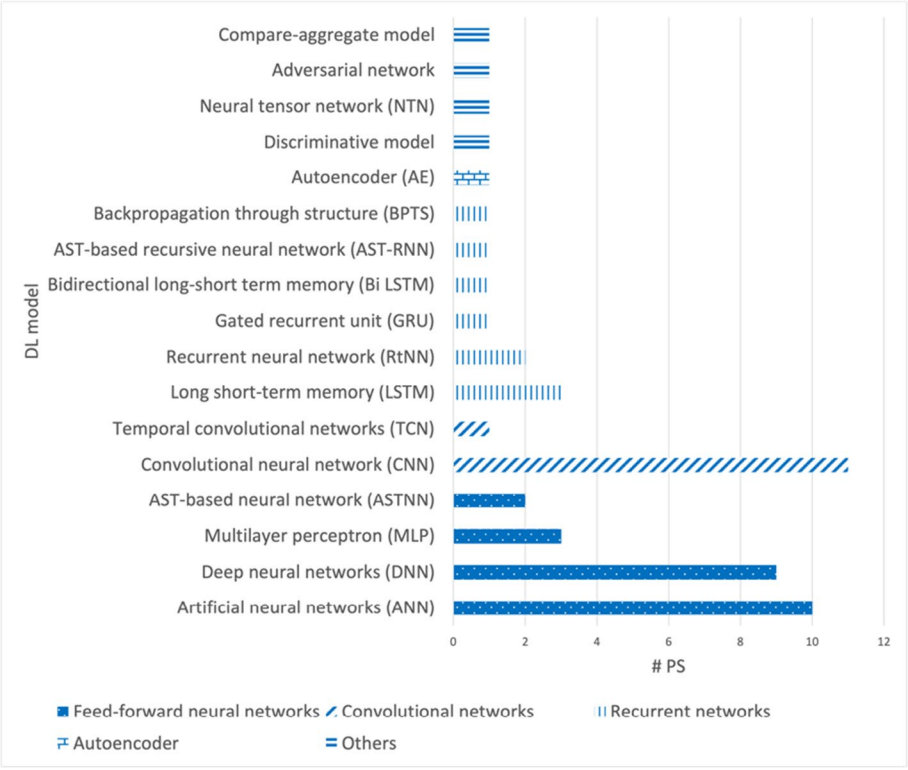


Fig. 7 DL models used for predictive classification

Table 11 DL models used for predictive classification

DL family	DL model	PS *	#PS
Feed-forward neural networks	Artificial neural networks (ANN)	[PS66, PS7, PS38, PS32, PS40, PS44, PS24, PS6, PS41, PS4]	10
	Deep neural networks (DNN)	[PS18, PS27, PS7, PS16, PS55, PS65, PS60, PS36, PS17]	9
	Multilayer perceptron (MLP)	[PS50, PS51, PS31]	3
	AST-based neural network (ASTNN)	[PS53, PS9]	2
Convolutional networks	Convolutional neural network (CNN)	[PS26, PS27, PS15, PS25, PS23, PS2, PS34, PS48, PS21, PS57, PS12]	11
	Temporal convolutional networks (TCN)	[PS64]	1
Recurrent networks	Long short-term memory (LSTM)	[PS27, PS25, PS49]	3
	Recurrent neural network (RtNN)	[PS23, PS19]	2
	Gated recurrent unit (GRU)	[PS25]	1
	Bidirectional long-short term memory (Bi LSTM)	[PS8]	1
	AST-based recursive neural network (AST-RNN)	[PS49]	1
	Backpropagation through structure (BPTS)	[PS28]	1
Autoencoder	Autoencoder (AE)	[PS23]	1
Others	Discriminative model	[PS1]	1
	Neural tensor network (NTN)	[PS43]	1
	Adversarial network	[PS56]	1
	Compare-aggregate model	[PS13]	1

*A PS may belong to one or more fields

used; hence, future studies may explore new models like generative Adversarial networks (GANs) and radial basis function networks (RBFN).

4.5.2 Which Deep Learning Models are used for Representation Learning?

Figure 8 and Table 12 present the studies that employed DL models for representation learning in bad smell detection. The recurrent network-based models were the most popular categories used to learn representation (38%). Convolutional networks (19%), autoencoder (18%), and attention networks (14%) were the second top categories. Neural (4%) and graph networks (4%) were not commonly used for representation learning. A total of 24 DL models were used, with the most popular models being the LSTM (9), Attention model (8), CNN (7), and RAE (6).

RQ5.2 summary. Recurrent network-based models were the most popular category used to learn representation (28%). A total of 24 DL models were used, with the most popular models being the LSTM (9), Attention model (8), CNN (7), and RAE (6). Future studies may explore new DL models for representation learning, such as the radial basis function network (RBFN).

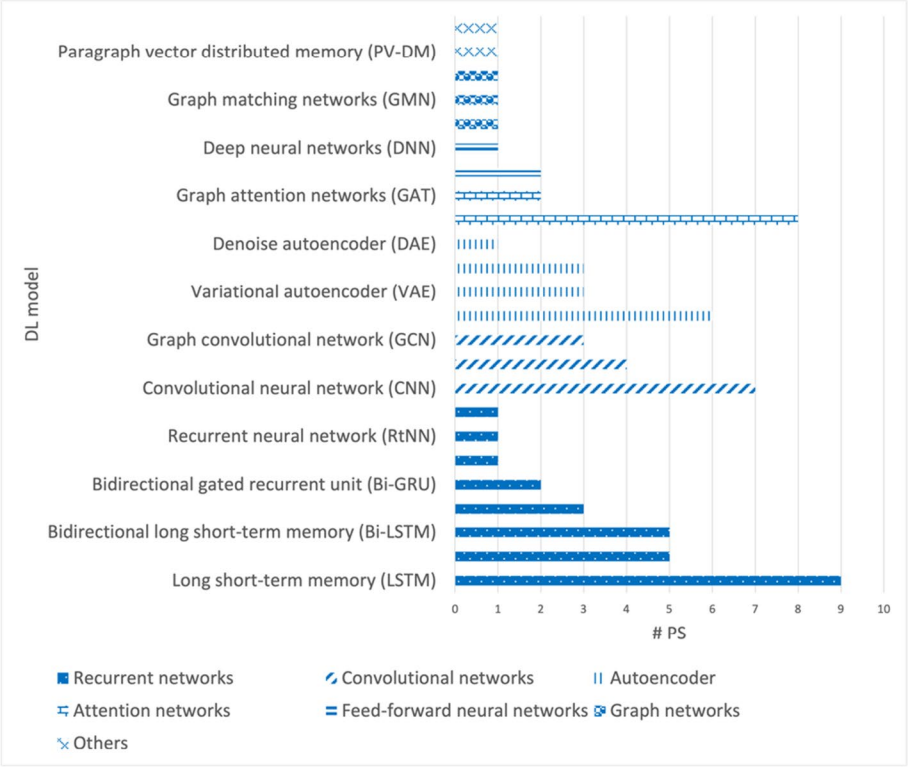


Fig. 8 DL models used for representation learning

Table 12 DL models used for representation learning

Category	DL model	PS *	#PS
Recurrent networks	Long short-term memory (LSTM)	[PS1, PS50, PS31, PS38, PS40, PS52, PS9, PS24, PS51]	9
	Recursive neural network (RvNN)	[PS53, PS52, PS9, PS28, PS20]	5
	Bidirectional long short-term memory (Bi-LSTM)	[PS1, PS35, PS13, PS61, PS41]	5
	AST-based long short-term memory (AST-LSTM)	[PS40, PS62, PS56]	3
	Bidirectional gated recurrent unit (Bi-GRU)	[PS53, PS9]	2
	Gated recurrent unit (GRU)	[PS58]	1
	Recurrent neural network (RtNN)	[PS28]	1
	Tree long short-term memory (T-LSTM)	[PS52]	1
Convolutional networks	Convolutional neural network (CNN)	[PS50, PS31, PS10, PS3, PS37, PS38, PS22]	7
	Tree-based convolutional neural network (TB-CNN)	[PS54, PS66, PS47, PS14]	4
	Graph convolutional network (GCN)	[PS67, PS42, PS40]	3
Autoencoder	Recursive autoencoders (RAE)	[PS16, PS29, PS39, PS42, PS60, PS20]	6
	Variational autoencoder (VAE)	[PS35, PS46, PS30]	3
	Autoencoder (AE)	[PS5, PS28, PS4]	3
	Denoise autoencoder (DAE)	[PS57]	1
Attention networks	Attention model	[PS1, PS50, PS47, PS43, PS38, PS40, PS67, PS51]	8
	Graph attention networks (GAT)	[PS59, PS51]	2
Feed-forward neural networks	Artificial neural networks (ANN)	[PS32, PS41]	2
	Deep neural networks (DNN)	[PS45]	1
Graph networks	Gated graph neural networks (GGNN)	[PS33]	1
	Graph matching networks (GMN)	[PS33]	1
	Graphsage	[PS43]	1
Others	Paragraph vector distributed memory (PV-DM)	[PS11]	1
	Skip gram model	[PS63]	1

*A PS may belong to one or more fields

Table 13 Methods used in each training hyperparameter

Training consideration	Values/methods (#PS)
Initialization	[Random (6), Xavier Initialization (2), Transfer Learning (1), Kaiming Uniform Method (1), Not specified (57)]
Learning rate	[0.001 (8), 0.0001 (3), 0.01 (3), 0.0005 (2), 0.005 (2), 0.002 (2), 0.015 (1), 0.00002 (1), 0.0002 (1), 0.25 (1), 0.05 (1), 0.003 (1), Not specified (43)]
Loss function	[Binary Cross Entropy (10), Cross Entropy (8), Eq. (5), Mean Squared Error (MSE) (4), User-defined (3), Relative Entropy (2), Euclidean Distance (2), Reconstruction Errors (1), Encode-Decode Loss (1), Adversarial Loss (1), Softmax Cross Entropy (1), Log Loss (1), Contrastive Loss Algorithm (1), Not specified (30)]
Optimization algorithm	[Adam (12), Stochastic Gradient Descent (6), Gradient Descent (4), Root Mean Square Prop (RMSPROP) (3), Adamax (2), Adagrad (2), Adadelta (1), 1BFGS (1), Nadam (1), Not specified (40)]
Overfitting handling	[dropout (13), regularization (7), ensemble methods (3), early stopping (2), Not specified (49)]

4.5.3 Which Training Hyperparameters are Considered in Building Deep Learning Models?

Various hyperparameters are involved in the DL model development due to the model's complexity. The considerations for the hyperparameter training are explored in this research question. Table 13 summarizes all hyperparameter values used in the selected PSs. Hyperparameters are certain parameters that affect the learning process. Those considered in this review (i.e., initialization, learning rate, loss function, optimization algorithms, and overfitting handling) affect the DL model training. Not all reviewed studies mentioned the considered training hyperparameters. Figure 9 shows the percentage of papers that stated the method used in each training parameter. Initialization was the least considered parameter, being mentioned in only 15% of the PSs. The loss function was the most considered training parameter, with almost half of the PSs (55%) stating the used loss function. Overfitting handling, learning rate, and optimization algorithm were considered in 27, 36, and 40% of the PSs, respectively.

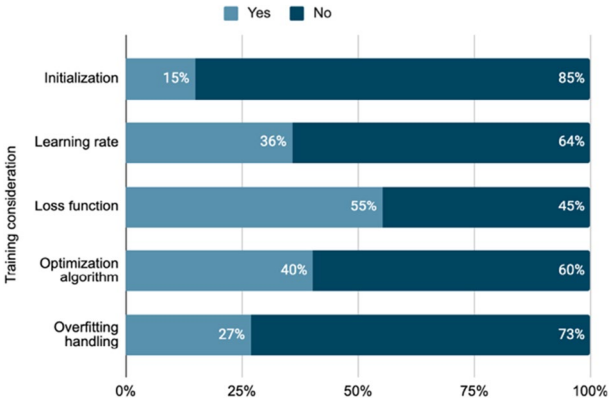


Fig. 9 PS distribution over the training considerations

Initialization is the process of assigning the neural network weights. The most commonly used method for weight initialization is random initialization (six papers) [PS14, PS18, PS24, PS33, PS40, and PS55]. Other initialization methods include Xavier initialization [PS6, PS21], transfer learning [PS23], and Kaiming uniform method [PS51].

Learning rate controls the weight adjustment rate of the neural network based on the loss function used. The learning rate used in the selected PSs spanned from 0.00002 to 0.25, with 0.001 being the most commonly used (eight papers) [PS10, PS13, PS32, PS33, PS44, PS47, PS52, and PS66].

Loss function is used to compute the error (i.e., the difference between the predicted and actual values). Binary cross-entropy is the most popular loss function used in the reviewed studies (10 papers) [PS1, PS50, PS27, PS31, PS13, PS9, PS34, PS60, PS41, and PS51]. Cross-entropy is the second top loss function (eight papers) [PS10, PS58, PS3, PS32, PS25, PS64, PS44, and PS28]. The remaining functions were only used in five or fewer studies. Finally, three studies defined their own loss functions [PS59, PS35, and PS40].

Optimization algorithms are used to find optimal parameters (i.e., weights and biases) by minimizing the loss function. The most frequently used algorithm is the Adam optimizer, which was utilized in 12 studies [PS1, PS47, PS10, PS33, PS13, PS52, PS64, PS44, PS34, PS21, PS41, and PS51]. The second most common algorithm is stochastic gradient descent (SGD), which was employed in six PSs [PS18, PS54, PS55, PS64, PS39, and PS11].

Overfitting handling is the method used to overcome the overfitting of DL models known to be susceptible to this issue. Different methods can be used to overcome overfitting. The dropout method was the most frequently used, with 13 studies considering usage [PS18, PS15, PS10, PS32, PS16, PS25, PS55, PS23, PS60, PS67, PS57, PS19, and PS41]. Regularization techniques, such as L1 and L2 regularization, which were used in seven PSs [PS26, PS32, PS39, PS23, PS67, PS57, and PS19], were the second most popular method. Ensemble methods [PS26, PS27, PS29] and early stopping techniques [PS64, PS23] were not common.

Most training hyperparameters were neglected in the reviewed studies, despite being fundamental aspects of the DL models. Although initialization has a significant impact on neural networks, it was the least considered parameter (Sutskever et al. 2013). We also emphasize herein the need to explore other sophisticated methods not used in the selected PSs. For example, the Restricted Boltzmann Machine is an initialization method that can be used in future studies. Conjugate gradient (CG) is an optimization algorithm that can help speed up the training of DL models (Le et al. 2011).

RQ5.3 summary. Initialization was the least considered parameter, being mentioned in only 15% of the PSs. The loss function was the most widely considered training parameter, with almost half of the PSs (55%) stating the usage of the function. Overfitting handling, learning rate, and optimization algorithm were considered in 27, 36, and 40% of the PSs, respectively. Other sophisticated methods for hyperparameter optimization that were not used in the selected PSs must be explored.

4.6 How are Deep Learning Models Evaluated and Implemented in Bad Smell Detection?

4.6.1 Which Validation Methods are used to Evaluate the Deep Learning Models?

The final phase of the DL model training (i.e., parameter learning phase) on the input data is the model validation in a real-world setting. Validation is a technique for assessing the performance of the trained model using previously unknown (i.e., unseen) data (Sammut and Webb 2011). This phase is

essential because it decreases overfitting, thereby increasing the generalization performance of DL models. It measures how well the model performed on previously unseen data. To achieve this goal, the dataset is usually split into two subsets: training and independent testing sets. When performing hyperparameter optimization, the testing set can no longer be considered an unbiased indicator of the model's generalization performance. Consequently, the training set is further separated into a third set known as a validation set used to select the optimum hyperparameter settings. Therefore, the testing set is used to objectively evaluate the model performance. Six validation methods were found in the primary studies briefly discussed below.

Train–test split. This category included studies performing a random split of the data into two sets: one used for training and another for testing. It also included studies that divided their data into training and testing sets and provided the number of instances used for training and testing without specifying the splitting ratio. The most common ratio was 80/20, where 80% of the data was used for training and 20% was utilized for testing [PS19, PS43, PS44, PS55, PS67, PS10]. The 70/30 followed, where 70% was used for training and 30% was used for testing [PS22, PS23, PS59]. One study considered 50/50 for training and testing [PS34], while other studies did not specify a ratio [PS3, PS6, PS8, PS17, PS21, PS35, PS36, PS39, PS42, PS57, PS61, PS66].

Train–valid–test split. This category included studies that split their data into three sets as follows: a training set for training a model; a validation set for hyperparameter optimization on unseen data; and a test set for evaluating the model performance. The most common split ratio was 80/10/10 for training/validation/testing [PS1, PS10, PS14, PS33, PS37, PS52, PS54]. The next most common ratio was 60/20/20 for training/validation/testing [PS9, PS13, PS40]. One study considered a 70/20/10 split [PS53], while other studies divided their data into three sets without specifying the ratio [PS24, PS51, PS65, PS45, PS49].

K-fold cross-validation. This method involves partitioning the data into k equally sized folds. We iterate over the data for k times. In each iteration, $k-1$ folds and onefold are used as the training and test sets, respectively. The model performance is calculated by averaging the performance results of the k iterations. The most frequent value of k used in the primary studies was $k=10$ [PS2, PS4, PS7, PS12, PS15, PS29, PS32, PS67], followed by $k=5$ [PS30, PS31, PS40]. One study considered $k=3$ [PS49]. Another considered $k=7$ [PS50].

Leave-one-out cross-validation. This method is a specification of the k-fold cross-validation, where $k=n$. In this validation method, all instances are used for training except for one instance is used for testing (*leave-one-sample-out*). Another variation of this method is to use all subjects for training and only one subject for testing, regardless of the number of instances belonging to each subject or system (*leave-one-subject-out*). Two studies used the *leave-one-sample-out* validation method [PS27, PS48], while three considered the *leave-one-subject-out* method [PS18, PS38, PS41].

Nested cross-validation. This method generates the train, validate, and test sets using two loops. The outer loop is used for error estimates, while the inner loop is employed for hyperparameter optimization. The inner loop divides the training set into two subsets: one is used for training, and another as a validation set. In this loop, the model is trained on the training set, and the hyperparameters minimizing the error on the validation set are selected. In the outer loop, the dataset is divided into many training and test sets. The errors are then averaged from each split to obtain a reliable model error estimate. This validation method was used in one study [PS64].

Table 14 shows the validation methods used in the selected primary studies. The train–test split (a.k.a., holdout) was the most commonly used validation method at 31%, followed by the train–valid–test split used in 23% of the PSs. Another common method found in the literature is the k-fold cross-validation representing 22% of the PSs. The leave-one-out (7%) and nested cross-validations (1%) were less popular. In the train–test and train–valid–test methods, the test set should be large, representative of the data, and not repeated in the training set.

Table 14 Validation methods used by the PSs

Validation method	PS*	#PS
Train-test	[PS35, PS61, PS17, PS36, PS19, PS57, PS22, PS44, PS43, PS39, PS55, PS66, PS59, PS10, PS3, PS42, PS21, PS6, PS34, PS8, PS67, PS23]	22
Train-valid-test	[PS9, PS24, PS40, PS49, PS13, PS52, PS53, PS37, PS45, PS65, PS10, PS33, PS51, PS14, PS54, PS1]	16
K-fold cross-validation	[PS2, PS30, PS31, PS32, PS15, PS40, PS49, PS29, PS50, PS12, PS67, PS4, PS7, PS58, PS26]	15
NA	[PS46, PS60, PS28, PS56, PS20, PS47, PS16, PS25, PS5, PS63, PS62, PS11]	12
Leave-one-out	[PS48, PS27, PS41, PS18, PS38]	5
Nested cross-validation	[PS64]	1

*A PS may belong to one or more fields

However, it is difficult for the testing set to have these features when the quantity of data available is limited. Hence, separating the data into various sets and only using a fraction for training can substantially impair the performance of data-hungry models, such as DL models. In these circumstances, cross-validation methods are preferred. However, cross-validation methods are time-consuming and require high computational power. This may be the reason behind the limited use of cross-validation methods. They take more time to train and evaluate ML models. Future studies might want to explore the stability of the DL model performance when various validation methods are used, along with the time and computational complexity required to help researchers select the most appropriate method when validating their DL models in bad smell detection.

RQ6.1 summary. Approximately 17% of the PS did not mention the use of any validation method. The train–test split is the most commonly used validation method, with 31% percentage, followed by the train–valid–test split used in 23% of the PSs. Future studies might want to explore the stability of the DL model performance when various validation methods are used, along with the required time and computational complexity.

4.6.2 Which Evaluation Metrics Have Been More Frequently Used to Report the Accuracy of Deep Learning Models?

Table 15 presents the types of evaluation metrics used by studies focusing on DL-based bad smell detection. Unsurprisingly, the majority of studies used metrics usually used in binary classification problems, such as precision, recall, F1-score, and accuracy. The top three evaluation metrics used were precision, recall, and F1-score, which corresponded to 85, 82, and 75% of the PSs, respectively. The accuracy measure used in 13% of the studies followed. Some of the less common evaluation metrics included AUC (9%), MCC (4%), and ROC (3%). The C-score, confidence score, false alarm rate, mean squared error (MSE), successrate@t and false positive rate were used only in 1% of the PSs. Even though most papers employed two or more evaluation metrics, few of them merely used one metric [PS2, PS3, PS12, PS15, PS28, PS34, PS36, PS49, PS64]. Four of these studies used accuracy; two studies used precision; one used the f-score; one used recall; and one used the AUC.

Most of the studies employed precision and recall metrics, which are inadequate for assessing the effectiveness of a model because they rely on an arbitrarily chosen threshold and are not suitable for data with imbalance classes (He and Garcia 2009). Using evaluation metrics that are threshold-independent (e.g., Brier score (Brier 1950) and MCC (Chicco and Jurman 2020)) is usually recommended for binary classification problems. Moreover, using evaluation metrics that are suitable in the case of imbalanced datasets (e.g., balanced accuracy and AUC (Ma and He 2013)) is highly suggested.

RQ6.2 summary. Recall (85%), precision (82%), and f-score (75%) are the most commonly used evaluation metrics. Bad smell detection is considered a binary classification problem with imbalanced classes; thus, future research may consider other metrics, such as AUC, MCC, and Brier.

4.6.3 Which Baseline Techniques are used to Evaluate the Deep Learning Models?

A new model or approach proposed to tackle an existing problem must be compared with other baseline techniques to show the superiority of the proposed model. Therefore, an important step when evaluating the DL model performance is to compare

Table 15 Evaluation metrics used by the PSs

Evaluation metric	PS*	#PS
Precision	[PS35, PS40, PS61, PS58, PS17, PS52, PS26, PS1, PS7, PS2, PS9, PS19, PS57, PS32, PS62, PS60, PS13, PS22, PS29, PS44, PS37, PS47, PS53, PS56, PS39, PS55, PS14, PS66, PS54, PS4, PS46, PS28, PS59, PS18, PS63, PS5, PS10, PS51, PS65, PS38, PS30, PS31, PS41, PS42, PS27, PS21, PS6, PS48, PS50, PS16, PS25, PS33, PS24, PS8, PS67, PS23, PS11]	57
Recall	[PS35, PS40, PS61, PS58, PS17, PS52, PS36, PS26, PS1, PS7, PS9, PS19, PS57, PS62, PS32, PS60, PS13, PS22, PS29, PS44, PS37, PS47, PS53, PS56, PS39, PS55, PS14, PS66, PS54, PS4, PS46, PS59, PS18, PS63, PS5, PS10, PS51, PS65, PS30, PS31, PS41, PS42, PS27, PS21, PS6, PS48, PS50, PS16, PS25, PS33, PS24, PS8, PS67, PS23, PS38]	55
F-score	[PS35, PS40, PS61, PS58, PS52, PS64, PS26, PS1, PS7, PS9, PS19, PS57, PS62, PS32, PS60, PS13, PS22, PS29, PS44, PS37, PS47, PS53, PS56, PS14, PS66, PS54, PS4, PS46, PS59, PS18, PS63, PS5, PS10, PS51, PS30, PS31, PS41, PS42, PS27, PS48, PS6, PS50, PS16, PS25, PS33, PS24, PS8, PS67, PS23, PS38]	50
Accuracy	[PS19, PS15, PS41, PS3, PS21, PS12, PS34, PS5, PS45]	9
Area under the curve	[PS7, PS49, PS27, PS45, PS23, PS38]	6
Matthew's correlation coefficient	[PS48, PS27, PS38]	3
Receiver operating characteristic curve	[PS48, PS39]	2
C-score	[PS17]	1
Confidence score	[PS11]	1
False alarm rate	[PS21]	1
Mean square error (MSE)	[PS43]	1
Successrate@t	[PS11]	1
False positive rate	[PS39]	1

*A PS may belong to one or more fields

the performance of the new model to that of available state-of-the-art models. In this research question, we investigated the baseline techniques used to compare the proposed DL models for bad smell detection. Comparisons are often gained by (1) re-implementing the models or (2) using the results from published models. The first approach is preferred, but due to the lack of source code availability (RQ6.4), some studies use the second approach. Nevertheless, the selected primary studies compared the proposed model with at least one of three baseline types: (1) other DL models used for bad smell detection; (2) other bad smell detection approaches (e.g., metrics-based); and (3) conventional ML models such as LR and SVM. Table 16 presents the most frequent baseline techniques used in at least two PSs. Most PSs compared the proposed DL model with at least one DL-based detection approach (51%). Approximately 31% of the PSs used other detection approaches as a baseline comparison. Few studies compared their DL model with ML-based detection approaches (13%). Only 5% of all PSs did not compare the performance of their DL models with any other approach. Among the various DL-based approaches, CDLH (H. Wei and Li 2017) was the most frequently used baseline (14 papers), employing a hashing function to optimize the similarity between two vector representations of the AST and the Tree-LSTM models to encode the features represented as ASTs. Deckard (Jiang et al. 2007) was the most widely used baseline approach (19 papers). It is a rule-based code clone detection approach that uses established rules to build characteristic vectors for each AST subtree, and then clusters them to detect the code clones. Sourcerercc (Sajnani et al. 2016) is another detection approach. It is a bag-of-words-based clone detection tool used as a baseline in 12 PSs. Among the list of ML-based approaches, the NB, RF, and SVM models were commonly used as the baselines. Two-thirds (39 papers out of 59) of the baselines that PSs used employed only one type of baseline, but considered more than one technique. One-third (20 papers out of 59) considered at least two baseline types. One paper used all three types of baselines [PS26]. Five used DL- + ML-based approaches [PS7, PS15, PS19, PS31, PS65]. Fourteen used DL-based + other approaches [PS32, PS38, PS39, PS41, PS43, PS44, PS50, PS52, PS57, PS59, PS62, PS63, PS66]).

Although most of the PSs compared their DL models with DL-based models, few studies considered conventional ML-based models. We recommend future studies to use both baseline types for comparison. Comparing DL models with conventional ML-based models is as important as comparing them with DL-based models because "the least complex model that is able to learn the target function is the one that should be implemented," as depicted in *Occam's Razor Principle* (Rasmussen and Ghahramani 2001).

RQ6.3 summary. Twelve percent of the studies did not report baseline comparisons. Approximately 66% compared their proposed model with the DL-based models, 46% compared theirs with other detection approaches, and 16% compared their models with the ML-based models. We recommend future studies to use ML- and DL-based baseline types for comparison.

4.6.4 Is the Source Code of Deep Learning Models Publicly Available?

Table 17 presents the source code availability along with the PSs. Only 20 studies (30%) made the source code available online. The other 47 papers (70%) did not publish the source code. Nevertheless, having access to the source code provides many benefits, including the following: (1) it saves time for future researchers proposing new DL models; (2) it improves the motivation to replicate a paper's findings; and (3) it facilitates the

Table 16 Baseline techniques used in the PSs

Baseline type	Baseline technique	PS*	#PS
DL-based detection approaches	CDLH	[PS39, PS60, PS66, PS56, PS52, PS14, PS67, PS32, PS16, PS54, PS33, PS44, PS40, PS9]	14
	DLC	[PS56, PS52, PS62, PS14, PS47, PS44, PS40]	7
	CNN	[PS61, PS52, PS19, PS53, PS23, PS64]	6
	Deepsim	[PS39, PS60, PS16, PS66, PS44, PS40]	6
	ASTNN	[PS67, PS58, PS33, PS44, PS13]	5
	Cclearner	[PS39, PS57, PS59, PS21, PS63]	5
	RtvNN	[PS66, PS67, PS32, PS58, PS33]	5
	DNN	[PS38, PS64, PS50, PS41]	4
	LSTM	[PS52, PS61, PS35, PS64]	4
	MLP	[PS31, PS15, PS19]	3
	Tree-LSTM	[PS52, PS35, PS13]	3
	Weighted-RAE	[PS60, PS16, PS67]	3
	Code2vec	[PS47, PS10]	2
	GGNN	[PS52, PS51]	2
	Oreo	[PS39, PS57]	2
	TBCCD	[PS37, PS51]	2
	TBCNN	[PS35, PS40]	2
	Word2vec	[PS10, PS22]	2
Other detection approaches	Deckard	[PS57, PS52, PS67, PS33, PS40, PS56, PS17, PS62, PS14, PS32, PS47, PS49, PS8, PS66, PS39, PS59, PS58, PS28, PS44]	19
	Sourcerercc	[PS57, PS56, PS17, PS52, PS62, PS14, PS67, PS47, PS8, PS44, PS40, PS55]	12
	Jdeodorant	[PS26, PS38, PS41, PS50, PS27]	5
	Nicad	[PS39, PS57, PS17, PS63, PS55]	5
	décor	[PS27, PS26]	2
	Jmove	[PS41, PS50]	2
	TACO	[PS27, PS30]	2
	CloneWorks	[PS39, PS55]	2
ML-based detection approaches	Naive Bayes (NB)	[PS7, PS15, PS25, PS12, PS48]	5

Table 16 (continued)

Baseline type	Baseline technique	PS*	#PS
	Random Forest (RF)	[PS12, PS7, PS48, PS19]	4
	Support Vector Machine (SVM)	[PS26, PS12, PS48, PS3]	4
	J48	[PS31, PS12, PS48]	3
	Decision Tree (DT)	[PS26, PS3]	2
	Logistic Regression (LR)	[PS65, PS3]	2
No basslines	-	[PS2, PS6, PS29, PS34, PS36, PS42, PS45, PS46]	8

*A PS may belong to one or more fields

Table 17 Source code availability

Source code availability	PS	#PS
No	[PS35, PS61, PS49, PS58, PS52, PS20, PS12, PS36, PS64, PS62, PS1, PS2, PS19, PS57, PS60, PS13, PS22, PS53, PS56, PS37, PS55, PS14, PS66, PS4, PS46, PS28, PS59, PS5, PS10, PS45, PS30, PS31, PS15, PS41, PS3, PS42, PS21, PS6, PS48, PS50, PS34, PS16, PS25, PS33, PS8, PS67, PS38]	47
Yes	[PS40, PS17, PS26, PS7, PS9, PS32, PS44, PS29, PS47, PS43, PS39, PS54, PS18, PS63, PS51, PS65, PS27, PS24, PS23, PS11]	20

baseline comparisons of DL models. Therefore, it is highly recommended to provide public access to the source code.

RQ6.4 summary. Only 20 studies (30%) made the source code available online; 47 papers (70%) did not publish the source code. Publishing the source code has many benefits; hence, it is highly recommended to make the source code available.

4.7 How are the Bad Smell Datasets Being Constructed and Used?

4.7.1 Which Systems and Datasets are Used in Deep Learning-based Bad Smell Detection?

Table 18 presents the main datasets and systems used in the selected studies and datasets/systems used in more than two studies. The overall number of datasets considered in the PSs was 148 datasets. Of the total, 81% (120 datasets) were used solely in one study, 8% (11 datasets) were used in two studies; and 11% (17 systems) were used in 28 studies. Figure 10 shows the number of datasets utilized in the PS distribution. Most PSs (64%) evaluated the DL models using one or two datasets. Only one study used 30 datasets. The remaining 23 studies used three to 13 datasets.

The table shows that BigCloneBench is the most prevalent dataset presented in 42% of the selected studies. Ojclone is the second most common dataset, being used in 18% of the PSs. The datasets collected from Github repositories and Google code jam were used in 9% of PSs each. Among the software systems used as a dataset, JEdit is the most prevalent system, representing 12% of the PSs. ArgoUML is the second most popular system used in 9% of the PSs. The rest of the datasets listed in the table were used in three to four PSs. The top two frequent datasets were used for code clone detection. The remaining datasets were employed in various bad smells, except for Google code jam and Atcoder, which were used solely for code clone detection.

Various datasets were used, and approximately 81% of these datasets were utilized only once. This denotes the lack of commonly accepted benchmarking datasets for bad smell detection studies. An exception is the code clone bad smell. Two benchmarking datasets existed and were widely used, explaining why the code clone is the most detected bad smell when using DL models. Moreover, the datasets presented in Table 18

Table 18 Data used by the PSs

Data used	PS*	#PS
Datasets		
BigCloneBench	[PS17, PS54, PS1, PS67, PS16, PS58, PS51, PS55, PS63, PS14, PS39, PS40, PS21, PS60, PS57, PS37, PS62, PS66, PS13, PS3, PS65, PS9, PS36, PS49, PS56, PS33, PS32, PS47]	28
Ojclone	[PS10, PS62, PS54, PS9, PS47, PS56, PS1, PS59, PS52, PS8, PS14, PS44]	12
GitHub repositories	[PS16, PS23, PS53, PS55, PS11, PS45]	6
Google code jam	[PS35, PS58, PS51, PS18, PS33, PS32]	6
Systems		
Jedit	[PS41, PS27, PS5, PS26, PS63, PS50, PS46, PS38]	8
Argouml	[PS5, PS28, PS22, PS48, PS26, PS34]	6
Junit	[PS41, PS48, PS27, PS50]	4
Weka	[PS41, PS42, PS27, PS50]	4
Pmd	[PS41, PS50, PS27, PS34]	4
Jhotdraw	[PS28, PS46, PS38]	3
Atcoder	[PS35, PS18, PS24]	3
Apache ant	[PS28, PS30, PS26]	3
Freeplane	[PS41, PS27, PS50]	3
Freemind	[PS38, PS46, PS34]	3
Eclipse	[PS5, PS22, PS48]	3
Areca	[PS41, PS27, PS50]	3
Apache lucene	[PS30, PS26, PS48]	3

*A PS may belong to one or more fields

have common characteristics: 1) they are created from open-source systems and are publicly available, and 2) they are long-lived systems and written in Java.

RQ7.1 summary. The total number of datasets considered in the PSs was 148 distinct datasets. Most PSs (64%) evaluated the DL models on one or two datasets. BigCloneBench (42%), Ojclone (18%), and JEdit (12%) were the most widely used datasets in the DL-based bad smell detection studies. Commonly accepted benchmarking datasets for bad smell detection are lacking.

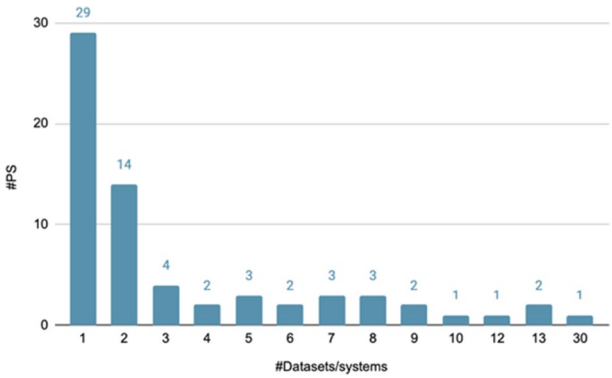


Fig. 10 Number of datasets used by the PSs

4.7.2 Which Programming Languages are Most Frequently Used to Construct Datasets?

Most datasets were constructed from systems written in Java, C, and C++. Table 19 shows the distribution of the programming languages considered in the selected primary studies. Java was the most commonly used programming language in 59 PSs, which may be due to the availability of open-source Java systems. C and C++ were the second and third most used programming languages (12 and 10 PSs, respectively). Other less common languages were C#, Python, Golang, and Scratch, which were used in four, four, one, and one PSs, respectively. One study detected bad smells at the model level [PS6]. Table 19 categorizes this study as programming language-independent (PL-indep).

Some primary studies explored cross-language bad smell detection and investigated the capability of DL models to detect bad smells across multiple programming languages. The studies that explored cross-language detection were [PS11] (C/C++, Java, Python, C#), [PS18] (Java, C#, Python), [PS23] (Java, C#), [PS24] (Java, Python), and [PS35] (Java, Python, C#, C/C++).

An interesting area of DL is transfer learning [PS23], where a DL model is trained on one programming language and then used for bad smell detection in other programming languages. This is beneficial, especially in the case of programming languages with low resources or datasets. It opens the door for designing tools that can be used universally, regardless of the system language.

RQ7.2 summary. Java was the most common programming language used in the 59 PSs, followed by C/C++ in 14 PSs. Datasets constructed from programming languages (e.g., C# and Python) are lacking.

4.7.3 Which Approaches Have Been Used to Create the Datasets?

Most studies tackled the problem of bad smell detection using the supervised learning approach (RQ3.2), which requires creating dataset oracles, which means labeling the

Table 19 Programming languages of the datasets used

Programing language	PS*	#PS
Java	[PS27, PS17, PS28, PS54, PS64, PS1, PS18, PS30, PS67, PS29, PS7, PS38, PS16, PS31, PS58, PS43, PS51, PS55, PS19, PS15, PS63, PS34, PS12, PS14, PS23, PS39, PS40, PS5, PS2, PS22, PS48, PS21, PS60, PS57, PS11, PS37, PS62, PS66, PS41, PS13, PS42, PS3, PS53, PS65, PS9, PS26, PS36, PS49, PS56, PS33, PS4, PS50, PS32, PS25, PS46, PS47, PS35, PS24, PS20]	59
C	[PS62, PS54, PS9, PS47, PS56, PS1, PS59, PS11, PS52, PS8, PS14, PS44]	12
C++	[PS10, PS35, PS62, PS54, PS9, PS47, PS52, PS8, PS14, PS44]	10
C#	[PS35, PS18, PS23, PS11]	4
Python	[PS35, PS18, PS24, PS11]	4
Golang	[PS45]	1
Scratch	[PS61]	1
PL-indep	[PS6]	1

*A PS may belong to one or more fields

extracted instances into smelly and non-smelly instances based on the bad smell being investigated. Accordingly, three main methods were used: (1) reference: using an existing dataset containing the labels from a reference; (2) automatic: using tools or rules for label creation; and (3) manual: manual inspection of the dataset and labeling them accordingly. Table 20 shows the approaches used by the primary studies. A total of 38 PSs used existing datasets; 21 papers automated the process, and 16 used the manual approach.

The labeling process automation was done using tools [PS7, PS23, PS42, PS48, PS53, PS55, PS64, PS65], similarity measure [PS8, PS10, PS18, PS24, PS35, PS43, PS45, PS52, PS61], semi-supervised labeling method [PS12], smell-introducing refactoring method [PS27], rules [PS34], or heuristics [PS65]. Most tools are used to detect code clone smells, such as Nicad [PS42, PS7], Sourcerercc [PS55, PS7], Ccfinderx [PS7], Cloneworks [PS7], Ctcompare [PS7], Deckard [PS7], Iclones [PS7], Simian [PS7], and Simcad [PS7]. Designite and Designite2 were utilized in [PS23, PS64] to extract the design smells, including complex methods, complex conditional, feature envy, and multifaceted abstraction.

The manual approach is usually combined with automatic and reference methods. Among the 61 studies, six merely used the manual method [PS28, PS2, PS11, PS50, PS6, PS29]; six first used an auto way to label the data and manually validated the extracted labels [PS42, PS23, PS64, PS65, PS48, PS7]; and four papers used an existing labeled dataset, but also manually validated the labels [PS57, PS63, PS3, PS40].

Most studies utilized an existing labeled dataset. Labeling is a time-consuming and labor-intensive process that might explain the extensive use of existing datasets and automatic approaches. Therefore, we highlight herein the need for publicly available labeled datasets and tools that can facilitate the dataset labeling process.

Various features were extracted and used (RQ4.1) to create the datasets. Table 21 presents the tools utilized by the selected primary studies for feature extraction. A broad range of tools can be used, but the table shows the tools used by at least two PSs. Twenty-four tools were found. Among them, 16 tools were used only once. Antlr (10 studies), Javalang (eight studies), and Pycparser (six studies) were the top three tools used for feature extraction. Antlr is mainly used to transform the source code written in C#, Python, and Java into abstract syntax trees (AST). Javalang is a python library used to extract the AST and tokenize the Java source code. Pycparser is used to generate the ASTs of the source code written in the C language. Java Development Tools (JDT) and Soot are used in four PSs. JDT is mainly used to extract method blocks and generate ASTs. Soot is used to extract the CFG.

The ratio of the PSs that made their dataset publicly available was only 30% (20 papers) compared to 70% (47) that did not publish their dataset or used a publicly available dataset without re-publishing it (Table 22). Appendix presents the publicly available datasets found in the primary studies. Most of the publicly available datasets were for the code clone smell, which was in alignment with the conclusion of the previous research questions. The code clone smell has many supported tools. Unlike other bad smells that need a careful inspection, labeling can easily be automated using similarity measures.

RQ7.3 summary. Most of the studies (38 PSs) used existing labeled datasets; 21 PSs automated the dataset labeling process, and 16 performed a manual inspection. Twenty-four tools were used for feature extraction. Among them, 16 tools were used only once. Antlr (10 PSs), Javalang (eight PSs), and Pycparser (six PSs) were the top three tools used for feature extraction. We highlight herein the need for publicly available labeled datasets and tools that can facilitate the dataset labeling process.

Table 20 Labeling methods used to create the oracles

Labeling method	PS*	#PS
Reference	[PS17, PS54, PS1, PS30, PS67, PS38, PS16, PS31, PS58, PS51, PS59, PS63, PS12, PS14, PS39, PS40, PS22, PS21, PS60, PS57, PS37, PS62, PS66, PS41, PS13, PS3, PS9, PS26, PS36, PS49, PS56, PS33, PS4, PS32, PS25, PS46, PS47, PS44]	38
Automatic	[PS10, PS61, PS27, PS64, PS18, PS8, PS7, PS43, PS55, PS19, PS34, PS12, PS23, PS48, PS24, PS42, PS35, PS53, PS65, PS45, PS52]	21
Manual	[PS42, PS23, PS3, PS40, PS28, PS2, PS64, PS48, PS65, PS57, PS63, PS11, PS50, PS6, PS29, PS7]	16

*A PS may belong to one or more fields

Table 21 Software tools used for feature extraction

Tools	PS *	#PS
Antlr	[PS16, PS17, PS28, PS40, PS21, PS60, PS18, PS63, PS61, PS8]	10
Javalang	[PS53, PS9, PS56, PS33, PS37, PS62, PS25, PS14]	8
Pycparser	[PS62, PS56, PS9, PS14, PS52, PS59]	6
Java Development Tools (JDT)	[PS15, PS28, PS27, PS49]	4
Soot	[PS43, PS40, PS51, PS58]	4
Ast2bin	[PS16, PS39]	2
Codesplit	[PS53, PS23]	2
Eclipse Astparser	[PS3, PS17]	2

*A PS may belong to one or more fields

5 Performance Analysis

We now analyze the DL model performance to answer RQ8. We have reported the performance for the dataset, bad smells, DL models, encoding techniques, preprocessing techniques, and features used. Synthesizing the data from several studies is difficult and many SLRs do not provide a synthesis (Cruzes and Dybå, 2011). This SLR followed the practices of Hall et al. (2012) and Hosseini et al. (2019) for data synthesis and analysis by combining and synthesizing our qualitative data. The main motivation behind using this approach is to generate a rich picture of the DL model performance for bad smell detection with respect to multiple factors. This was achieved by using boxplots depicting the DL model performance for various factors, such as dataset, features, bad smells, and encoding techniques. The boxplots contained the detection performance of three evaluation metrics: precision, recall, and f-score. We selected these measures because they were the most used evaluation metrics in the PSs. The number of studies shown in each boxplot was different from that reported in the previous section. This was because some studies used evaluation metrics other than precision, recall, and f-score; hence, they were excluded from the boxplots. For instance, the Bigclonebench dataset was used in 28 studies (Table 18). However, in Fig. 11, the number of studies that used Bigclonebench was only 27 ($N=27$) because

Table 22 Dataset availability

Dataset availability	PS	#PS
No	[PS10, PS61, PS17, PS28, PS20, PS1, PS30, PS67, PS8, PS38, PS16, PS31, PS58, PS55, PS19, PS15, PS52, PS59, PS63, PS34, PS6, PS12, PS5, PS2, PS22, PS21, PS60, PS57, PS37, PS62, PS66, PS41, PS13, PS35, PS42, PS3, PS53, PS36, PS49, PS56, PS33, PS4, PS45, PS50, PS25, PS46, PS47]	47
Yes	[PS27, PS64, PS54, PS18, PS29, PS7, PS43, PS51, PS14, PS23, PS39, PS40, PS48, PS11, PS24, PS65, PS9, PS26, PS32, PS44]	20

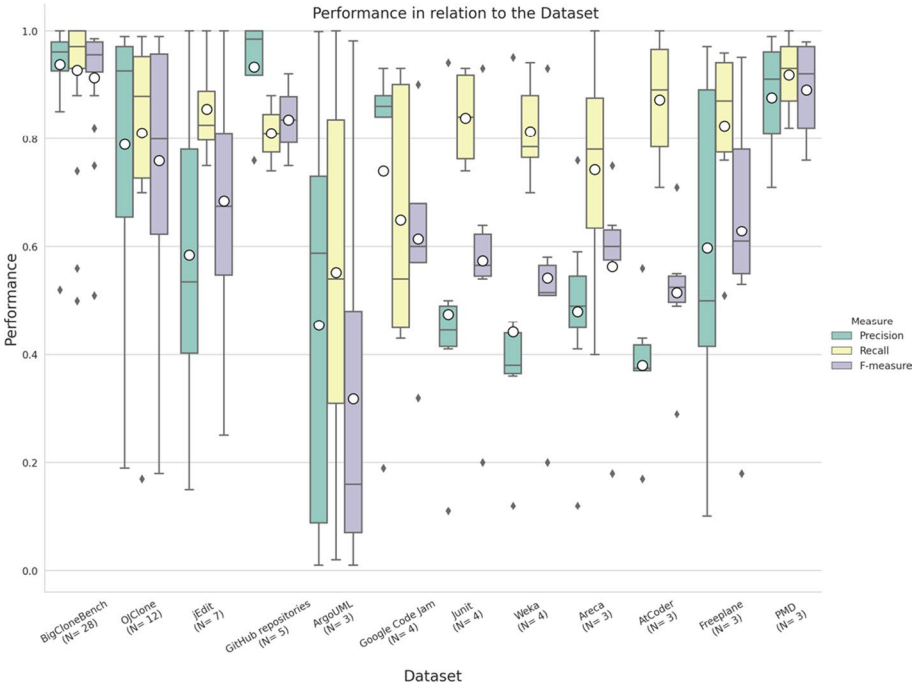


Fig. 11 Detection performance with respect to the datasets

they did not report the detection performance in terms of precision, recall, and f-score in [PS36]. Moreover, data were only included in each boxplot if at least three PSs utilized the same technique/feature/dataset (i.e., N must be greater than or equal to 3). Finally, we only considered the best performance if a study reported the results of multiple configurations. It is worth mentioning that the performance analysis does not show strengths and weaknesses of the DL models as each PS conducted its experiment on a context that cannot be generalized to another context, rather it shows the current state of DL models performance in a rich picture, and research can be inspired by this analysis to explore new techniques to improve the detection performance.

5.1 Dataset

Figure 11 presents the detection performance relative to the dataset used. We observed a wide variety of reported model performances while using the Github repositories because they were usually collected from a wide range of systems. The Bigclonebench dataset showed the least variation in the detection performance and the best precision, recall, and F1 scores. BigCloneBench is a clone detection benchmark dataset that satisfies the benchmarking criteria such as 1) representative (it has all types of code clones), 2) experimentally verified (it has been evaluated), and 3) reusability (it is publicly available). There is a need for the development of high-quality benchmark bad smells datasets to be used in training and evaluating the DL model. The figure also illustrates that building models for

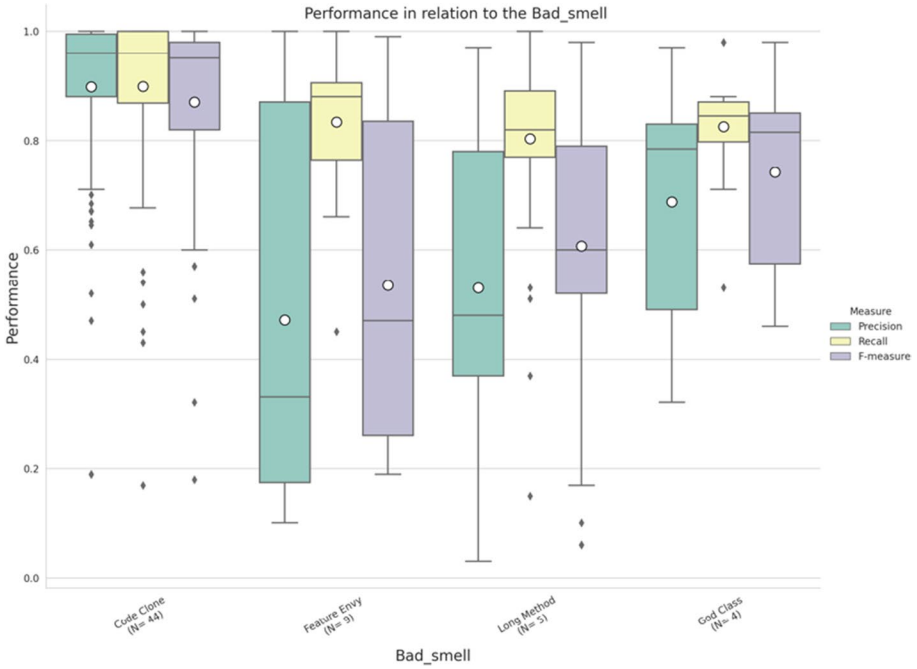


Fig. 12 Detection performance with respect to the bad smells

certain systems may be more challenging than for others. Most datasets achieved higher recall scores compared to precision scores.

5.2 Bad Smells

Figure 12 depicts the detection performance of the most used bad smells. The code clone smell achieved the highest detection performance compared to the other smells, which might be due to the large interest of the research community in code clone smell detection. Accordingly, new techniques were proposed to improve the detection performance. Feature Envy, Long Method, and God Class have achieved similar detection performances in terms of recall and nearly similar precision scores. Therefore, there is a need for new DL models, features, or encoding techniques to enhance the detection of performance of Feature Envy, Long Method, and God Class code smells.

5.3 Deep Learning Models

Figures 13 and 14 show the detection performances of the DL models used for predictive classification and representation learning for bad smell detection, respectively. Figure 13 depicts the similar recall scores for the DL models. The precision scores varied between the models and the F1 scores. The CNN and ANN models showed the lowest precision scores. Moreover, most of the DL models had higher or nearly equal recall scores compared to their precision scores. An exception to that is the DNN models, which showed higher precision scores.

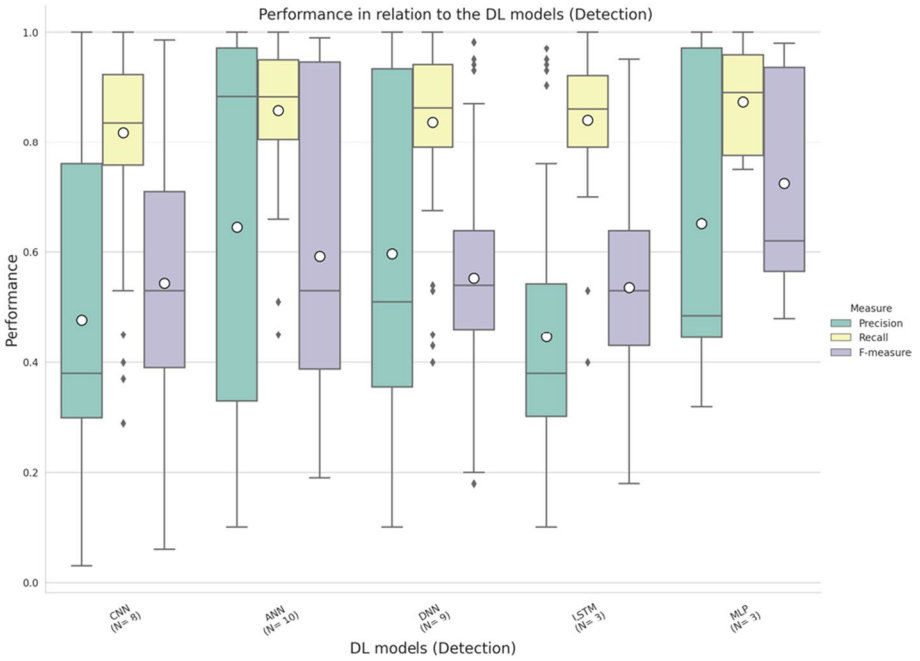


Fig. 13 Detection performance with respect to the DL models for predictive classification

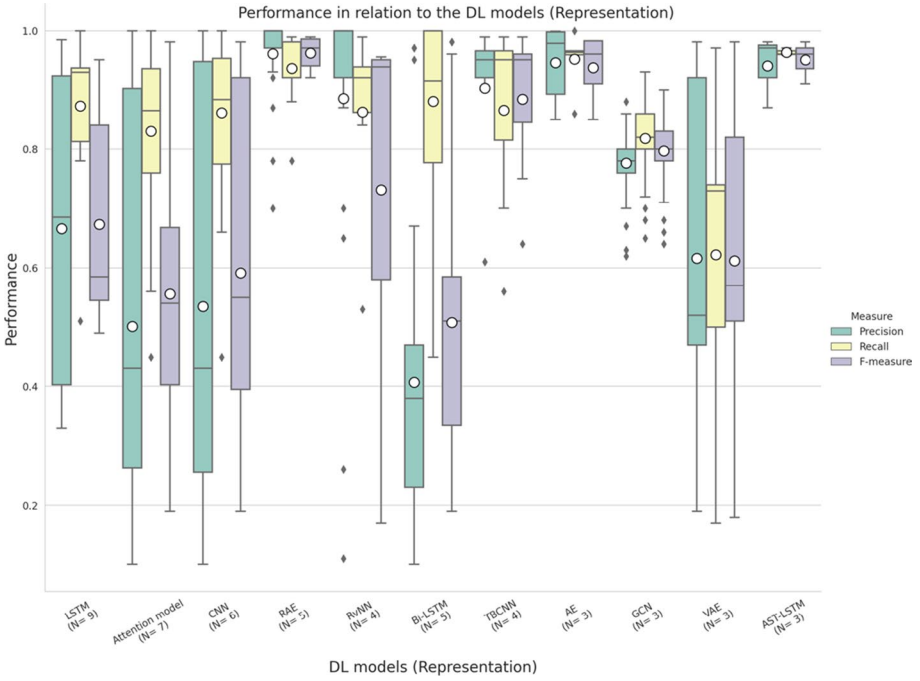


Fig. 14 Detection performance with respect to the DL models for representation learning

Among the DL models used for representation learning (Fig. 14), RAE- and VAE-based models performed well. The TBCNN and AST-based LSTM models achieved high detection performances. The LSTM and the Bi-LSTM did not perform well in bad smell detection. RAE and VAE, which are autoencoder variations, is a good technique commonly used for representation learning (Sarker 2021). The TBCNN is a CNN variation specially designed to encode the source code ASTs. The AST-based LSTM is a variation of the well-known LSTM model adapted to work with AST. This explains the good performance of these two models.

The CNN and the LSTM are well suited and very effective in computer vision and NLP, respectively (Jaiswal et al. 2021). Using these models for bad smell detection must be further explored to enhance the detection performance. A general observation is that the DL models specifically designed to encode features for bad smell detection (e.g., TBCNN and AST-based LSTM) achieve higher scores; more experiments are needed to show the superiority of these models over other DL models.

5.4 Encoding Techniques

Figure 15 presents the encoding techniques used in the PSs with respect to the detection performance. We show only studies that reported the use of either Word2vec or TF-IDF, along with those that did not use any encoding technique. The figure suggests that the detection performance of the DL models is not related to the employed encoding technique. The studies that did not employ any encoding techniques had nearly similar scores to word2vec encoding. TF-IDF showed less performance variation due to the low number of the PSs that used it (three studies) compared to the word2vec (23 studies) and no

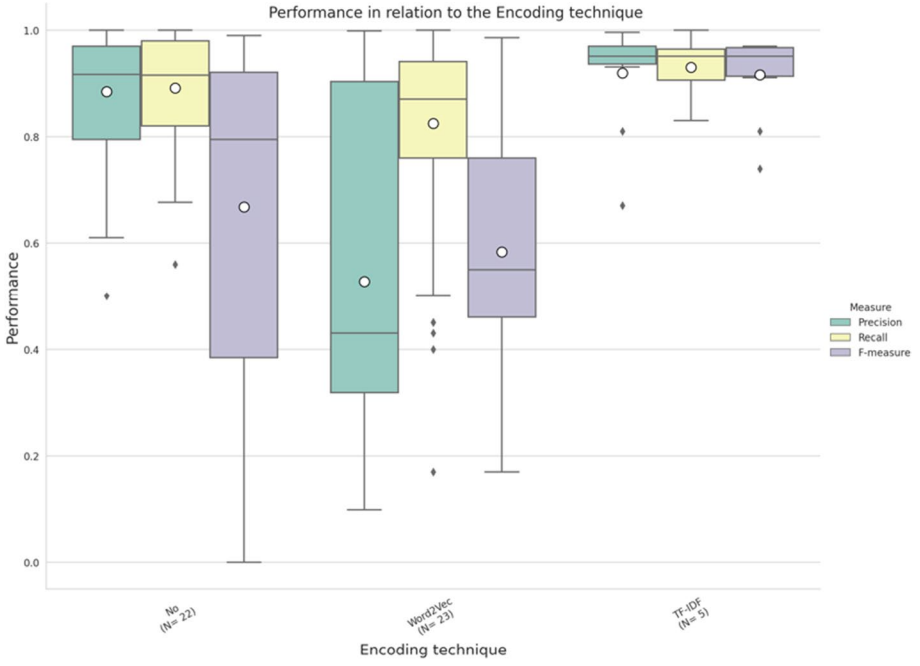


Fig. 15 Detection performance with respect to the encoding techniques

(22 studies) groups. Most of the studies used DL for representation learning, replacing the need for encoding. This did not dramatically affect the DL model performance.

5.5 Preprocessing Techniques

Figure 16 illustrates the preprocessing techniques used in the PSs with respect to the detection performance. The figure shows that studies that utilized filtering and tokenization have less performance variation compared to feature selection, class imbalance, and transformation techniques. Studies that have employed normalization techniques have the highest scores with minimal variation in terms of precision and recall metrics.

5.6 Features

Figure 17 illustrates the detection performance relative to the features used in the PSs. We observe that using software metrics and statement trees as features have achieved higher performance scores compared to textual information. Future studies may explore other techniques that can be used to capture features extracted from textual data. A small performance variation was observed between models using other features (i.e., AST, source code, and CFG).

RQ8 summary. Many factors can affect the DL model performance, but the DL model selection has the highest impact. The DNN, autoencoders (RAE and VAE), TBCNN, and AST-based LSTM showed the best detection performances. Moreover, building DL models for certain systems may be more challenging than for others. The code clone smell achieved the highest detection performance among all smells. The DL model performance

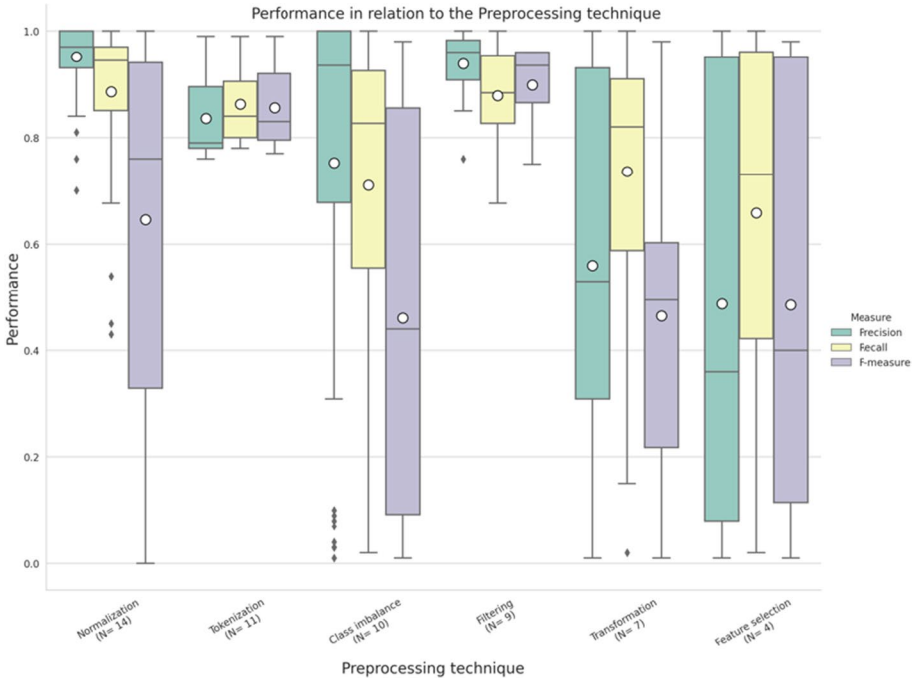


Fig. 16 Detection performance with respect to the preprocessing techniques

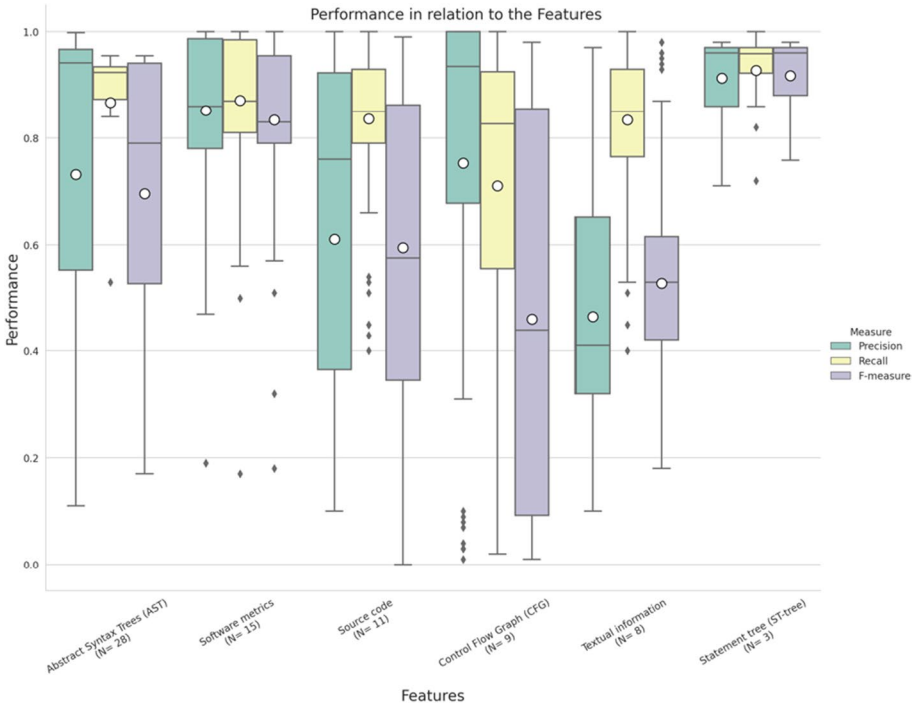


Fig. 17 Detection performance with respect to the features

for bad smell detection is not related to the encoding technique used. Finally, a small performance variation was observed between models with respect to the features used. We highly suggest that future studies investigate the effectiveness of each factor on the DL model performance. This will help in deriving solid conclusions about the factors that are most influential and need to be considered when building DL models.

6 Discussion and Implications

This section discusses how DL-based bad smell detection may be further explored to drive toward a more accurate detection performance. We summarize the main findings here and pinpoint open issues and opportunities for future research.

6.1 Exploring Other Bad Smells (RQ2)

Although 49 types of bad smells were detected using DL techniques, Code Clone was the most detected code smell in the PSs. This smell type might have been frequently detected owing to the following reasons: 1) availability of large-scale benchmarking labeled datasets; 2) availability of tools that can detect this smell type; 3) low cost of building a new labeled dataset; 4) availability of tools; and 5) nature of this bad smell, which facilitates the labeling process.

In terms of DL utilization for bad smell detection, we observed that most of the PSs focused on one smell type. Other bad smell types are also known to negatively influence the software quality (Olbrich et al. 2010), and they are found to frequently occur in software systems (Zhang et al. 2011). A research gap exists in the detection of bad smells other than the Code Clone smell using DL techniques. Therefore, we recommend further research to exploit and fill this gap. The Feature Envy, Data Class, God Class, and Long Method bad smells are frequently found in software systems (Zhang et al. 2011), and they greatly affect the software quality (Olbrich et al. 2010); however, very few studies have explored the usage of DL models to detect these smells. The capability of the Bi-LSTM, RtNN, RvNN, GCN, and GMN DL models has never been explored for detecting the Data Class, God Class, and Long Method bad smells, which can be investigated in future research.

6.2 Model-Level Detection (RQ2)

Bad smell detection at the early stages of the software development cycle reduces the maintenance cost. However, most of the PSs detected these smells at the code level. Only one study investigated design smell detection at the model level [PS6]. Despite the usefulness of detecting bad smells at the model level, it has received little attention in the context of DL-based detection.

In the future, researchers should investigate design smell detection by employing DL techniques. DL models should also be developed to detect bad smells at the model level using UML diagrams such as class and use case diagrams.

6.3 Exploring New Learning Approaches (RQ3)

Existing studies have tackled the problem of bad smell detection as a supervised learning problem. The supervised learning approach requires a huge number of labeled datasets; however, as discussed in RQ7, such datasets are lacking. The issue of lacking labels might be addressed by exploring other learning approaches. One learning approach that might tackle this problem is self-supervised learning, which has been used in other areas like natural language processing (Devlin et al. 2019) and computer vision (Ohri and Kumar 2021) and has shown promising results. The use of self-supervised learning for DL-based bad smell detection was investigated in one study [PS47]. However, self-supervised learning is worth further investigation. Future research may explore new pretext tasks directly derived from the source code rather than the AST. The self-supervised learning model scalability in various down-streaming tasks may also be investigated.

Another interesting research area to explore is transfer learning, which has been explored solely in one study [PS23]. They used this learning technique to detect bad smells across multiple programming languages. This learning approach is useful because most of the available datasets had been extracted from systems written in Java. The use of transfer learning is promising but requires more research. Future studies can investigate how to transfer the knowledge extracted from Java systems to other programming languages that lack datasets, such as Python, C#, and Golang. Moreover, researchers can also investigate the feasibility of transferring knowledge from one task (e.g., code clone detection) to another (e.g., Feature Envy, God Class, and Long Method detection).

6.4 From Binary Classification to Multilabel (RQ3)

Current DL models output a single final prediction that indicates a smelly or a non-smelly instance. This is known as binary classification. The current architecture implies the need for a single DL model for each bad smell type. Most of the investigated studies used binary classification, and very few PSs used multi-classification [PS3, PS13, PS17, PS55, PS57], in which they did not solely predict whether or not a code fragment was a clone but also predicted the clone type. Nevertheless, multi-classification is useful in detecting multiple types of bad smells and can further be explored in the future.

However, using multi-classification remains an issue because a single instance may be affected by two smells at the same time. For example, a class can be a God Class and a Data Class. In this example, multi-classification is not enough. Multi-label classification permits the assignment of multiple labels to each instance. A recent study explored code smell detection using multi-label datasets and ML models (Guggulothu and Moiz 2020). Future research must consider utilizing DL models to detect various smell types in a single model and recognize if an instance has two smell types. A possible method to approach this is by defining new activation functions other than the ReLU and Sigmoid functions at the output layer of a DL model (e.g., using softmax in the last neural network layer to return the probability of each class).

6.5 Exploitation of Hyperparameter Optimization (RQ4)

Most of the studies did not report the hyperparameters used in the DL models. Most of them also used default hyperparameters and did not consider hyperparameter optimization or a manual configuration of a set of hyperparameters. One study [PS64] performed a detailed hyperparameter optimization to optimize the learning rate, activation function, optimization algorithm, and the number of layers through Sequential Bayesian Model-based Optimization (SBMO). We emphasize herein the importance of the hyperparameter optimization of DL models because the development of DL models is an iterative and empirical process, and no one-size-fits-all hyperparameter configuration can be used. We recommend that the hyperparameter optimization for DL models in bad smell detection research should be considered as required rather than a choice. Manual configuration is also time-consuming. Future studies should utilize systematic optimization techniques like grid search, differential evolution, random search, and genetic algorithm (Tantithamthavorn et al. 2019).

6.6 Data Preprocessing (RQ5)

Although data preprocessing is essential and can influence the DL model performance, many PSs did not explore most of the preprocessing techniques. Imbalanced labels and feature selection were utilized in a few PSs. A research gap exists in the usage of data preprocessing methods, such as data transformation, tokenization, data normalization, and filtering. These preprocessing methods have been explored, but many have not yet been investigated in detail. For instance, log transformation, square root transformation, Z-score normalization, and decimal scaling are data transformation and normalization techniques that could be explored. The principal component analysis can also be used as a feature selection technique. Hence, more research on the use of data preprocessing

techniques for DL-based bad smell detection is needed to fill the gap. Future research can investigate the effectiveness of these techniques on both the detection performance and the computational cost as well.

6.7 More Attention on Model Evaluation (RQ6)

The use of multiple metrics for the DL model evaluation is important. Most of the PSs used at least two performance metrics, but most did not include the training time in the proposed DL model evaluation. Complex models may require additional resource consumption (e.g., storage and computational time cost). The model selection should consider all these aspects (Lim et al. 2000). Therefore, we highly recommend that the training time be considered as an evaluation metric along with the performance metrics in DL-based bad smell detection.

Although most of the PSs compared their DL models with DL-based models, few studies considered conventional ML models. Comparing DL models with conventional ML models is as important as comparing them with DL-based models because, as stated by the *Occam's Razor Principle* (Rasmussen & Ghahramani 2001), a simple model should be selected over a complex model when both can learn the target function. One study (Fakhoury et al. 2018) applied the principle to the DL models. Surprisingly, the DL model failed to pass Occam's Razor Test. Future studies must use the two baseline types for comparative comparisons, that is, comparing the new DL models with previous and well-known DL models (e.g., CNN and LSTM) and with traditional ML models (e.g., DT and SVM).

6.8 Automated Tools (RQ6)

We observed that existing DL models proposed for bad smell detection were not implemented in a software tool. In addition, the DL model's source code was not made publicly available. Automating the detection model can help researchers compare new models to state-of-the-art ones and assist in constructing new datasets considering that building datasets from scratch is costly and time-consuming. Only a few studies implemented their DL models into a software tool. For example, [PS16] developed a web application that can detect code clones by uploading code fragments. A command-line tool was implemented in [PS7] to detect code clones. In [PS45], their detection model for the code clone was implemented as a cloud-based web application and a command-line tool.

Future research on DL-based bad smell detection may consider implementing their DL models into automated tools. We suggest that they use more efficient methods other than the command line and web applications. Future works can also implement their detection models as plugin tools that can be used within existing IDEs, such as Eclipse. The use of autonomous bots for the automatic support of software activities is becoming popular in software engineering (Storey and Zagalsky 2016). We recommend that future studies implement their detection model using software bots that can facilitate the detection process of bad smells.

6.9 Large-Scale and Publicly Available Datasets (RQ7)

Our SLR shows a lack of labeled bad smell datasets. Although we are not the first to highlight this issue, we emphasize herein the importance of datasets in the era of DL-based models because DL models are known to be data-hungry. Large-scale datasets are required to build robust DL models. Therefore, future research must consider constructing bad smell

datasets and making them publicly available to researchers. Most publicly available datasets are for the code clone smell. Therefore, future studies may want to focus on constructing datasets for other bad smell types, such as Feature Envy, God Class, Long Method, and Data Class. Constructing a dataset for various programming languages (e.g., Python, C, and C#) other than Java is also necessary because they are popular programming languages used by many companies. Finally, the labeling procedures used so far are not unified. Future studies may propose more systematic methods to label bad smells. For example, semi-supervised learning can be utilized to construct labeled datasets (Sheneamer et al. 2017).

6.10 Deep Learning Model Performance (RQ8)

The performance analysis performed in this SLR did not indicate that the investigated factors are necessarily affecting the DL model performance but rather provided insights about possible factors. We highly suggest that future studies investigate the effectiveness of each factor on the DL model performance. This will help derive solid conclusions about the factors that are most influential and need to be considered when in DL model construction.

7 Threats to the Validity

SLRs are susceptible to various threats that may influence the findings and conclusions of a study (Zhou et al. 2016). Therefore, potential threats must be identified. The validity threats observed during this SLR are outlined below.

7.1 Construct Validity

Construct validity refers to the connection between the research hypothesis and the results. Threats that belong to this category are typically research questions, databases used, and search terms.

Search Bias We selected six digital libraries that are well-known in computer science and software engineering. Covering all databases and venues is impractical. We further applied the snowballing procedure to mitigate the risk of missing related papers. We believe that by following these procedures, the most relevant studies were included in this SLR.

Incomplete Search Term Constructing a comprehensive and informative search string is not trivial. A missing term might affect the search result. In the context of DL, most studies do not necessarily include the term “deep learning” in the title, abstract, or keywords. They alternatively use the technique name (e.g., LSTM or CNN). Therefore, we included the DL technique names to ensure the search string comprehensiveness. A pilot search was also performed with various search strings to mitigate the risk of missing important terms. Moreover, synonyms and spelling variations were considered in the search string construction.

7.2 Internal Validity

Internal validity is concerned with the relation between treatment (i.e., set of primary studies included) and outcome. The possible threats related to this category are quality assessment and data extraction.

Subjective Quality Assessment The answers to some of the quality assessment questions are not decisive; thus, a subjective assessment might be given to a study based on how the evaluator interprets it. To mitigate this threat, one author conducted the quality assessment for all studies, and the other two verified the assessment by applying the quality assessment to all studies. The results were then compared. Any disagreement was discussed for resolution.

Bias in Data Extraction We created a data extraction form to extract relevant data from the primary studies. A pilot data extraction was performed on 10 studies to ensure that the data extraction items were clear and reflective of the targeted research questions. The data extraction form was then refined. To mitigate the risk of data extraction bias, one author extracted the data, and the other two verified the extracted data of all studies. The data were compared, and any disagreements were discussed and resolved.

7.3 External Validity

External validity refers to the extent to which the study findings can be generalized. The generalizability and the time span are two issues related to this validity.

Primary Study Generalizability The outcome of this study cannot be generalized to other domains. The findings of this SLR might be inaccurate for a detection system relying on DL (i.e., different DL models might yield better detection performances than those analyzed in this SLR when applied to other domains).

Restricted Time Span Researchers are usually unable to include related papers outside the time frame specified in the planning phase. DL is a fast-growing area, and new DL techniques are introduced every day. Consequently, a certain DL-based bad smell detection research that succeeded in producing a high detection performance may not have been included because it was not published at the time of writing.

7.4 Conclusion Validity

Conclusion validity refers to threats that influence the ability to draw the same conclusions if the procedure is repeated. The main issue related to this category is the selection of included studies.

Bias in Study Selection The selection of the primary studies might be subjective based on the researcher's judgment. The selection process was performed in two rounds to mitigate this threat. The first round was based on the title and the abstract of the papers. The second round was based on a full-text read. Another researcher was consulted if a decision cannot be made on a particular study. Moreover, a detailed description of the selection process was described, including the research questions, search string, search process, and exclusion and inclusion criteria. The questions related to the quality assessment and the data extraction form were provided.

8 Conclusion

This SLR mainly aimed to summarize and synthesize the studies that used DL techniques for bad smell detection. In this review, we identified the bad smell types, DL techniques, features used as inputs, evaluation methods of the DL models, and datasets and systems used in the DL model training. This study also aimed to explore the detection performance of DL models in relation to other factors (i.e., datasets, features, encoding techniques, preprocessing techniques, and bad smell types). A comprehensive literature review was conducted to attain the study's objectives and answer the research questions. A total of 67 studies were selected by following a systematic approach and analyzing the inclusion/exclusion criteria and the quality of the studies. Apart from the previously mentioned recommendations and the future research opportunities, the key findings from this SLR are given below based on the research questions:

Bad Smells Bad smells under the code smell category are the most addressed smells. Among them, Code Clone, Feature Envy, and Long Method are the most recurring smells. God Class is the most studied design smell. Most primary studies detected bad smells at the code level, while a few detected it at the model level. Most studies detected bad smells at the method level.

Framework The primary objective of using DL models is to explicitly learn the representation of the input feature. Typically, the DL model output is a feature vector. Supervised learning is the most adopted learning approach, followed by the semi-supervised and unsupervised learning approaches.

DL Models Most of the DL models were constructed using only one feature type. Graph-based features are the most common features (e.g., ASTs). Data-related issues are not considered in most of the studies. CNN, ANN, and DNN are the most popular DL models used to build prediction classification models, while the LSTM, Attention model, CNN, and RAE are the most popular DL models used for representation learning. The training hyperparameters are usually neglected.

Evaluation The train–test split is the most frequently used validation method. Recall, precision, and f-score are the most widely used evaluation metrics. The majority of the studies compared the proposed model with the DL-based ones, while some compared the proposed model with the ML-based models. Tensorflow and Keras are the top two frameworks used for DL model implementation. The source code is usually not publicly available.

Datasets The majority of the studies evaluated the DL models using one or two datasets. The BigCloneBench, Ojclone, and JEdit datasets are the most widely used datasets in DL-based bad smell detection studies. Java is the most common programming language used for dataset construction. Three main methods are used for dataset labeling: 1) using an existing dataset containing labels from a reference; 2) automatically using tools or rules to create labels; and 3) manually inspecting the dataset and labeling them accordingly. Most studies used the first approach. The majority of the datasets are not publicly available, and the publicly available datasets are only for the Code Clone smell.

The DL model performance in bad smell detection is influenced by many factors. Specifically, the detection performance of the DL models is associated with the datasets used and the DL modeling techniques. Although most of the DL models achieved a comparative detection performance in terms of recall, most could only achieve low precision. DL-based bad smell detection still remains a challenge, and performance improvements are needed.

Appendix

Table 23 The selected primary studies

Study ID	Ref	Title
PS1	(Meng and Liu 2020)	A deep learning Approach for a source code detection model using self-attention
PS2	(Li et al. 2020b)	A Deep Learning Based Approach to Detect Code Clones
PS3	(Kim 2020)	A Deep Neural Network-Based Approach to Finding Similar Code Segments
PS4	(Hadj-Kacem and Bouassida 2018)	A Hybrid Approach To Detect Code Smells using Deep Learning
PS5	(Patnaik and Padhy 2021)	A Hybrid Approach to Identify Code Smell Using Machine Learning Algorithms
PS6	(Sidhu et al. 2020)	A machine learning approach to software model refactoring
PS7	(Mostaeen et al. 2020)	A machine learning based framework for code clone validation
PS8	(Dong et al. 2020)	A Novel Code Stylometry-based Code Clone Detection Strategy
PS9	(Zhang et al. 2019)	A Novel Neural Source Code Representation Based on Abstract Syntax Tree
PS10	(Xie et al. 2020)	A Source Code Similarity Based on Siamese Neural Network
PS11	(Nafi et al. 2020)	A universal cross language software similarity detector for open source software categorization
PS12	(Sheneamer et al. 2021)	An Effective Semantic Code Clone Detection Framework using Pairwise Feature Fusion
PS13	(Liang and Ai 2021)	AST-path Based Compare-Aggregate Network for Code Clone Detection
PS14	(Chen et al. 2019)	Capturing Source Code Semantics via Tree-based Convolution over API-enhanced AST
PS15	(Sheneamer 2018)	CCDLC Detection Framework-Combining Clustering with Deep Learning Classification for Semantic Clones
PS16	(Zhang and Wang 2021)	CCEyes An Effective Tool for Code Clone Detection on Large-Scale Open Source Repositories
PS17	(Li et al. 2017a)	CCLearner A Deep Learning-Based Clone Detection Approach
PS18	(Nafi et al. 2019)	CLCDSA Cross Language Code Clone Detection using Syntactical Features and API Documentation
PS19	(Ullah et al. 2021)	Clone detection in 5G-enabled social IoT system using graph semantics and deep learning model
PS20	(Xue et al. 2018)	Clone-Slicer- Detecting Domain Specific Binary Code Clones through Program Slicing
PS21	(Dişlî and Tosun 2020)	Code Clone Detection with Convolutional Neural Networks
PS22	(Wu and Wang 2021)	Code Similarity Detection Based on Siamese Network
PS23	(Sharma et al. 2021)	Code smell detection by deep direct-learning and transfer-learning
PS24	(Perez and Chiba 2019)	Cross-language clone detection by learning over abstract syntax trees
PS25	(Hamdy and Tazy 2020)	Deep hybrid features for code smells detection
PS26	(Barbez et al. 2019)	Deep Learning Anti-Patterns from Code Metrics History

Table 23 (continued)

Study ID	Ref	Title
PS27	(H. Liu et al. 2019)	Deep Learning Based Code Smell Detection
PS28	(White et al. 2016)	Deep Learning Code Fragments for Code Clone Detection
PS29	(Tufano et al. 2018)	Deep Learning Similarities from Different Representations of Source Code
PS30	(Hadj-Kacem and Bouassida 2019b)	Deep Representation Learning for Code Smells Detection using Variational Auto-Encoder
PS31	(Guo et al. 2019)	Deep semantic-Based Feature Envy Identification
PS32	(Zhao and Huang 2018)	DeepSim- Deep Learning Code Functional Similarity
PS33	(Wang et al. 2020a, b, c)	Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree
PS34	(Das et al. 2019)	Detecting Code Smells using Deep Learning
PS35	(Zhang et al. 2021)	Disentangled Code Representation Learning for Multiple Programming Languages
PS36	(Kim 2019)	Enhancing code clone detection using control flow graphs
PS37	(Kaur and Saini 2021)	Enhancing the Software Clone Detection in BigCloneBench- A Neural Network Approach
PS38	(Ren et al. 2021)	Exploiting Multi-aspect Interactions for God Class Detection with Dataset Fine-tuning
PS39	(Zeng et al. 2019)	Fast Code Clone Detection Based on Weighted Recursive Autoencoders
PS40	(Hua et al. 2021)	FCCA Hybrid Code Representation for Functional Clone Detection Using Attention Networks
PS41	(Wang et al. 2020a)	Feature Envy Detection based on Bi-LSTM with Self-Attention Mechanism
PS42	(Yuan et al. 2020)	From Local to Global Semantic Clone Detection
PS43	(Nair et al. 2020)	funcGNN- A Graph Neural Network Approach to Program Similarity
PS44	(Fang et al. 2020)	Functional Code Clone Detection with Syntax and Semantics Fusion Learning
PS45	(Wang et al. 2019)	Go-Clone- Graph-Embedding Based Clone Detector for Golang
PS46	(Hadj-Kacem and Bouassida 2019a)	Improving the Identification of Code Smells by Combining Structural and Semantic Information
PS47	(Bui et al. 2021)	InferCode Self-Supervised Learning of Code Representations by Predicting Subtrees
PS48	(Fakhoury et al. 2018)	Keep it simple Is deep learning good for linguistic smell detection-
PS49	(Buch and Andrzejak 2019)	Learning-Based Recursive Aggregation of Abstract Syntax Trees for Code Clone Detection
PS50	(Yin et al. 2021)	Local and Global Feature Based Explainable Feature Envy Detection
PS51	(Mehrotra et al. 2021)	Modeling Functional Similarity in Source Code with Graph-Based Siamese Networks
PS52	(Wang et al. 2020c)	Modular Tree Network for Source Code Representation Learning
PS53	(Xu 2021)	Multi-Granularity Code Smell Detection using Deep Learning Method based on Abstract Syntax Tree

Table 23 (continued)

Study ID	Ref	Title
PS54	(Yu et al. 2019)	Neural Detection of Semantic Code Clones via Tree-Based Convolution
PS55	(Saini et al. 2018)	Oreo- Detection of Clones in the Twilight Zone
PS56	(Wei and Li 2018)	Positive and Unlabeled Learning for Detecting Software Functional Clones with Adversarial Training
PS57	(Guo et al. 2020)	Review Sharing via Deep Semi-Supervised Code Clone Detection
PS58	(Wu et al. 2020)	SCDetector- Software Functional Clone Detection Based on Semantic Tokens Analysis
PS59	(Li et al. 2020a)	Semantic code clone detection via event embedding tree and gat network
PS60	(Feng et al. 2020)	Sia-RAE A Siamese Network based on Recursive AutoEncoder for Effective Clone Detection
PS61	(Zhang et al. 2020)	Siamese-Based BiLSTM Network for Scratch Source Code Similarity Measuring
PS62	(Wei and Li 2017)	Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code
PS63	(Gao et al. 2019)	Teccd- A tree embedding approach for code clone detection
PS64	(Ardimento et al. 2021)	Temporal convolutional networks for just-in-time design smells prediction using fine-grained software metrics
PS65	(Saini et al. 2019)	Towards Automating Precision Studies of Clone Detectors
PS66	(Jo et al. 2021)	Two-Pass Technique for Clone Detection and Type Classification Using Tree-Based Convolution Neural Network
PS67	(Ji et al. 2021)	Code Clone Detection with Hierarchical Attentive Graph Embedding

Table 24 The number of primary studies published in each venue and the publication type

Venue	Publication Type	#PS
IEEE/ACM international conference on automated software engineering (ASE)	C	3
Journal of systems and software	J	3
IEEE access	J	3
International conference on software analysis evolution and reengineering (SANER)	C	3
IEEE/ACM international conference on software engineering (ICSE)	C	2
ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering (ESEC/FSE)	C	2
International joint conference on artificial intelligence (IJCAI)	C	2
International joint conference on neural networks (IJCNN)	C	2
IEEE international conference on software maintenance and evolution (ICSME)	C	2
IEEE international conference on information communication and software engineering (ICICSE)	C	2
IEEE/ACM international conference on mining software repositories (MSR)	C	2
IEEE Transactions on software engineering	J	2
International journal of open source software and processes (IJOSSP)	J	2
International wireless communications and mobile computing (IWCMC)	J	2
ACM sigsoft international symposium on software testing and analysis	C	2
Applied sciences	J	2
International journal of machine learning and cybernetics	J	1
International conference on program comprehension (ICPC)	C	1
International journal of software engineering and knowledge engineering	J	1
Journal of theoretical and applied information technology	J	1
International journal of electrical computer engineering	J	1
International journal of computers and applications	J	1
International conference on software quality reliability and security (QRS)	C	1
International conference on software maintenance and evolution (ICSME)	C	1
International conference on software engineering and knowledge engineering (SEKE)	C	1
International conference on software engineering (ICSE)	J	1
Neurocomputing	J	1
ACM/IEEE international symposium on empirical software engineering and measurement (ESEM)	C	1
International conference on intelligent computing and human computer interaction (ICHCI)	C	1

Table 24 (continued)

Venue	Publication Type	#PS
Findings of the association for computational linguistics (ACL-IJCNLP)	C	1
ACM transactions on software engineering and methodology (TOSEM)	J	1
Annual computers software and applications conference (COMPSAC)	C	1
Asia pacific software engineering conference (APSEC)	C	1
Asia pacific symposium on internetware	C	1
Bili im teknolojileri dergisi (journal of information technologies)	J	1
Complexity journal	J	1
Evaluation of Novel Approaches to Software Engineering (ENASE)	C	1
IEEE annual computers software and applications conference (COMPSAC)	C	1
International conference on dependable systems and their applications (DSA)	C	1
IEEE international conference on machine learning and applications	C	1
IEEE intl conf on parallel distributed processing with applications big data cloud computing sustainable computing communications social computing networking (ISPA BDCLOUD SOCIALCOM SUSTAINCOM)	C	1
IEEE region 10 conference (TENCON)	C	1
IEEE transactions on reliability	J	1
ACM international conference on computing frontiers	C	1
IEICE transactions on information and systems	J	1
Ininternational conference on neural information processing	C	1
Workshop on forming an ecosystem around software transformation	W	1

J: Journal, C: Conference, W: Workshop

Table 25 Types of bad smells detected using DL

Category	Bad smell	DL models *	#PS
Code smells	Code clone	Class: GGNN (1): [PS33], GMN (1): [PS33], RAE (1): [PS29] Code fragments: GAT (1): [PS59], VAE (1): [PS35], LSTM (1): [PS35], CNN (1): [PS22], ANN (1): [PS7], DNN (1): [PS7] File: TBCNN (3): [PS14, PS47, PS54], LSTM (2): [PS1, PS52], AST-LSTM (2): [PS62, PS56], Attention model (2): [PS1, PS47], RvNN (2): [PS28, PS52], T-LSTM (1): [PS52], Bi-LSTM (1): [PS1], Discriminative model (1): [PS1], Adversarial network (1): [PS56], RtNN (1): [PS28], BPTS (1): [PS28], AE (1): [PS28] Method: CNN (8): [PS2, PS15, PS57, PS37, PS10, PS12, PS21, PS3], DNN (8): [PS55, PS17, PS16, PS36, PS18, PS65, PS60, PS45], LSTM (6): [PS1, PS24, PS40, PS51, PS9, PS49], ANN (5): [PS24, PS32, PS44, PS40, PS66], RAE (6): [PS42, PS16, PS39, PS60, PS20, PS29], Attention model (6): [PS1, PS67, PS43, PS40, PS51, PS47], Bi-LSTM (3): [PS1, PS8, PS13], RvNN (3): [PS9, PS20, PS28], TBCNN (4): [PS66, PS47, PS14, PS54], AST-LSTM (3): [PS62, PS40, PS56], GCN (3): [PS40, PS67, PS42], GAT (1): [PS51], RtNN (1): [PS28], AE (2): [PS5, PS28], AST-RNN (1): [PS49], ASTNN (1): [PS9], BPTS (1): [PS28], Bi-GRU (1): [PS9], Compare-Aggregate Model (1): [PS13], DAE (1): [PS57], GRU (1): [PS58], GraphSAGE (1): [PS43], MLP (1): [PS51], NTN (1): [PS43], Adversarial network (1): [PS56], Discriminative model (1): [PS1], Skip-gram model (1): [PS63] Project: RtNN (1): [PS19], LSTM (1): [PS61], PV-DM (1): [PS11]	58
	Feature envy	Class: ASTNN (1): [PS53], Bi-GRU (1): [PS53], RvNN (1): [PS53], CNN (1): [PS23] RtNN (1): [PS23], AE (1): [PS23] Method: CNN (3): [PS50, PS31, PS27], LSTM (2): [PS50, PS3], ANN (2): [PS4, PS41], MLP (2): [PS50, PS31], AE (1): [PS4], VAE (2): [PS46, PS30], Bi-LSTM (1): [PS41], Attention model (1): [PS50]	9
	Long method	Method: VAE (2): [PS46, PS30], AE (2): [PS4, PS5], ANN (1): [PS4], DNN (1): [PS27]	5
	Large class	Class: DNN (1): [PS27], LSTM (1): [PS27], AE (1): [PS5]	2
	Data class	Class: AE (1): [PS4], ANN (1): [PS4]	1
	Misplaced class	Class: CNN (1): [PS27]	1
	Brain class	Class: CNN (1): [PS34]	1
	Brain method	Method: CNN (1): [PS34]	1
	Complex conditional	Method: AE (1): [PS23], CNN (1): [PS23], RtNN (1): [PS23]	1
	Complex method	Method: AE (1): [PS23], CNN (1): [PS23], RtNN (1): [PS23]	1
	Empty catch block	Method: ASTNN (1): [PS53], Bi-GRU (1): [PS53], RvNN (1): [PS53]	1
	Attribute name and type are opposite	Attribute: CNN (1): [PS48]	1
	Attribute signature and comment are opposite	Attribute: CNN (1): [PS48]	1

Table 25 (continued)

Category	Bad smell	DL models *	#PS
	Expecting but not getting a collection	Method: CNN (1): [PS48]	1
	Expecting but not getting a single instance	Method: CNN (1): [PS48]	1
	Get method does not return	Method: CNN (1): [PS48]	1
	Get method does not return corresponding attribute	Method: CNN (1): [PS48]	1
	Get more than an accessor	Method: CNN (1): [PS48]	1
	Is returns more than a Boolean	Method: CNN (1): [PS48]	1
	Method name and return type are opposite	Method: CNN (1): [PS48]	1
	Method signature and comment are opposite	Method: CNN (1): [PS48]	1
	Name suggests Boolean but type does not	Attribute: CNN (1): [PS48]	1
	Not answered question	Method: CNN (1): [PS48]	1
	Not implemented condition	Method: CNN (1): [PS48]	1
	Says many but contains one	Attribute: CNN (1): [PS48]	1
	Says one but contains many	Attribute: CNN (1): [PS48]	1
	Set method returns	Method: CNN (1): [PS48]	1
	Transform method does not return	Method: CNN (1): [PS48]	1
	Validation method does not confirm	Method: CNN (1): [PS48]	1
Design smells	God class	Class: CNN (3): [PS25, PS38, PS26], LSTM (2): [PS25, PS38], ANN (2): [PS4, PS38], AE (1): [PS4], GRU (1): [PS25], Attention model (1): [PS38]	4
	Blob	Class: VAE (2): [PS46, PS30]	2
	Deficient encapsulation	Class: ASTNN (1): [PS53], Bi-GRU (1): [PS53], RvNN (1): [PS53], TCN (1): [PS64]	2
	Insufficient modularization	Class: ASTNN (1): [PS53], Bi-GRU (1): [PS53], RvNN (1): [PS53], TCN (1): [PS64]	2
	Multifaceted abstraction	Class: TCN (1): [PS64], CNN (1): [PS23], AE (1): [PS23], RtNN (1): [PS23]	2
	Broken modularization	Class: TCN (1): [PS64]	1
	Cyclically-dependent Modularization	Class: TCN (1): [PS64]	1
	Cyclic hierarchy	Class: TCN (1): [PS64]	1
	Deep hierarchy	Class: TCN (1): [PS64]	1
	Functional decomposition	Class: ANN (1): [PS6]	1
	Hub-like modularization	Class: TCN (1): [PS64]	1
	Missing hierarchy	Class: TCN (1): [PS64]	1

Table 25 (continued)

Category	Bad smell	DL models *	#PS
	Multipath hierarchy	Class: TCN (1): [PS64]	1
	Rebellious hierarchy	Class: TCN (1): [PS64]	1
	Unexploited encapsulation	Class: TCN (1): [PS64]	1
	Unutilized abstraction	Class: TCN (1): [PS64]	1
	Wide hierarchy	Class: TCN (1): [PS64]	1
	Broken hierarchy	Class: TCN (1): [PS64]	1
	Imperative abstraction	Class: TCN (1): [PS64]	1
	Unnecessary abstraction	Class: TCN (1): [PS64]	1

ANN: Artificial neural networks, AST-LSTM: AST-based long short-term memory, ASTNN: AST-based neural network, AST-RNN: AST-based recursive neural network, AE: Autoencoder, BPTS: Backpropagation through structure, Bi-GRU: Bidirectional gated recurrent unit, Bi-LSTM: Bidirectional long short-term memory, Bi LSTM: Bidirectional long-short term memory, CNN: Convolutional neural network, DNN: Deep neural networks, DAE: Denoise autoencoder, GGNN: Gated graph neural networks, GRU: Gated recurrent unit, GAT: Graph attention networks, GCN: Graph convolutional network, GMN: Graph matching networks, LSTM: Long short-term memory, MLP: Multilayer perceptron, NTN: Neural tensor network, PV-DM: Paragraph vector distributed memory, RtNN: Recurrent neural network, RAE: Recursive autoencoders, RvNN: Recursive neural network, TCN: Temporal convolutional networks, T-LSTM: Tree long short-term memory, TB-CNN: Tree-based convolutional neural network, VAE: Variational autoencoder.

*A PS may belong to one or more fields

Table 26 Publicly available datasets for each bad smell

PS	Bad smells	Link
PS7	Code clone	https://github.com/pseudoPixels/CloneCognition
PS9	Code clone	https://github.com/zhangj1994/astnn
PS11	Code clone	https://github.com/Kawser-nerd/CroLSim
PS14	Code clone	https://github.com/milkfan/TBCAA
PS18	Code clone	https://github.com/Kawser-nerd/CLCDSA
PS23	CM, CC, FE, MA	Sharma, T., 2021. tushartushar/DeepLearningSmells: public release. Zenodo, https://doi.org/10.5281/zenodo.4571626
PS24	Code clone	https://www.csg.ci.i.u-tokyo.ac.jp/projects/clone/
PS26	God Class	https://github.com/antoineBarbez/CAME/
PS27	FE, LM, LC, MC	https://github.com/liuhuigmail/DeepSmellDetection
PS29	Code clone	https://sites.google.com/view/learningcodesimilarities
PS32	Code clone	https://github.com/parasol-aser/deepsim
PS39	Code clone	https://github.com/zyj183247166/Recursive_autoencoder_xiaojie
PS40	Code clone	https://github.com/preesee/CodeCloneDetection
PS43	Code clone	https://github.com/aravi11/funcGNN
PS44	Code clone	https://github.com/shiyy123/FCDetector
PS48	Linguistic smells	https://github.com/Smfakhoury/SANER-2018-KeepItSimple-
PS51	Code clone	https://drive.google.com/drive/folders/1x6ePcgJVtFVOycENEiYBpz5FYzsx2D-?usp=sharing
PS54	Code clone	http://github.com/yh1105/datasetforTBCCD
PS64	IA, MA, UA, UNA, DE, UE, BM,CDM, IM,HLM,BH, CH, DH, MH, MPH, RH, WH	https://bit.ly/3oItvpN
PS65	Code clone	https://github.com/Mondego/InspectorCloneArtifacts

Complex method (CM), Complex conditional (CC), Feature envy (FE), Long method (LM),Large class (LC), Misplaced class (MC), Imperative Abstraction (IA), Multifaceted Abstraction (MA), Unnecessary Abstraction (UA), Unutilized Abstraction (UNA), Deficient Encapsulation (DE), Unexploited Encapsulation (UE), Broken Modularization (BM), Cyclic D. Modularization (CDM), Insufficient Modularization (IM), Hub-like Modularization (HLM), Broken Hierarchy (BH), Cyclic Hierarchy (CH), Deep Hierarchy (DH), Missing Hierarchy (MH), Multipath Hierarchy (MPH), Rebellious Hierarchy (RH), Wide Hierarchy (WH).

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

Data Availability The datasets generated during and/or analyzed during the current study are available in the GitHub repository, https://github.com/amalazba/Deep-Learning-Approaches-for-Bad-Smell-Detection-SLR.

Declarations

Conflicts of Interests/Competing Interests The authors have no conflicts of interest to declare that are relevant to the content of this article.

References

AbuHassan A, Alshayeb M, Ghouti L (2021) Software smell detection techniques: A systematic literature review. J Softw Evol Process 33(3):e2320. https://doi.org/10.1002/smr.2320

Alkharabsheh K, Crespo Y, Manso E, Taboada JA (2019) Software Design Smell Detection: A systematic mapping study. Software Qual J 27(3):1069–1148. https://doi.org/10.1007/s11219-018-9424-8

Al-Shaaby A, Aljamaan H, Alshayeb M (2020) Bad Smell Detection Using Machine Learning Techniques: A Systematic Literature Review. Arab J Sci Eng. https://doi.org/10.1007/s13369-019-04311-w

Anne-Wil Harzing (2006) Publish or perish. Harzing.Com. Retrieved January 23, 2022, from https://harzing.com/resources/publish-or-perish

Ardimento P, Aversano L, Bernardi ML, Cimitile M, Iammarino M (2021) Temporal convolutional networks for just-in-time design smells prediction using fine-grained software metrics. Neurocomputing 463:454–471. https://doi.org/10.1016/j.neucom.2021.08.010

Azeem MI, Palomba F, Shi L, Wang Q (2019) Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. Inf Softw Technol 108:115–138. https://doi.org/10.1016/j.infsof.2018.12.009

Barbez A, Khomh F, Gueheneuc Y-G (2019) Deep Learning Anti-Patterns from Code Metrics History. IEEE International Conference on Software Maintenance and Evolution (ICSME) 2019:114–124. https://doi.org/10.1109/ICSME.2019.00021

Bengio Y, Courville AC, Vincent P (2013) Representation Learning: A Review and New Perspectives. IEEE Trans Pattern Anal Mach Intell 35(8):1798–1828. https://doi.org/10.1109/TPAMI.2013.50

Brier G (1950). VERIFICATION OF FORECASTS EXPRESSED IN TERMS OF PROBABILITY. https://doi.org/10.1175/1520-0493(1950)078%3c0001:VOFEIT%3e2.0.CO;2

Brown WH, Malveau RC, McCormick HWS, Mowbray TJ (1998) AntiPatterns: refactoring software, architectures, and projects in crisis (1st edn). John Wiley & Sons, Inc.

Buch, L, Andrzejak, A (2019) Learning-Based Recursive Aggregation of Abstract Syntax Trees for Code Clone Detection. 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), 95–104. https://doi.org/10.1109/SANER.2019.8668039

Bui, NDQ, Yu, Y, Jiang, L (2021) InferCode: Self-Supervised Learning of Code Representations by Predicting Subtrees. 2021 IEEE/ACM 43rd International Conference on Software Engineering (*ICSE)*, 1186–1197. https://doi.org/10.1109/ICSE43902.2021.00109

Caram FL, Rodrigues BRDO, Campanelli AS, Parreiras FS (2019) Machine Learning Techniques for Code Smells Detection: A Systematic Mapping Study. Int J Software Eng Knowl Eng 29(02):285–316. https://doi.org/10.1142/S021819401950013X

Chen, L, Ye, W, Zhang, S (2019) Capturing source code semantics via tree-based convolution over API-enhanced AST. Proceedings of the 16th ACM International Conference on Computing Frontiers, 174–182. https://doi.org/10.1145/3310273.3321560

Chicco D, Jurman G (2020) The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation. BMC Genomics 21:6. https://doi.org/10.1186/s12864-019-6413-7

Cruzes DS, Dybå T (2011) Research synthesis in software engineering. Inf Softw Technol 53(5):440–455. https://doi.org/10.1016/j.infsof.2011.01.004

Das, AK, Yadav, S, Dhal, S (2019) Detecting Code Smells using Deep Learning. TENCON 2019 - 2019 IEEE Region 10 Conference (TENCON), 2081–2086. https://doi.org/10.1109/TENCON.2019.8929628

Devlin J, Chang M-W, Lee K, Toutanova K (2019) BERT: Pre-training of deep bidirectional transformers for language understanding. http://arxiv.org/abs/1810.04805. Accessed 07-03-2022

Dìşlì H, Tosun A (2020) Code Clone Detection with Convolutional Neural Networks. Bilişim Teknolojileri Dergisi 13(1):1–12. https://doi.org/10.17671/gazibtd.541476

Dong W, Feng Z, Wei H, Luo H (2020) A Novel Code Stylometry-based Code Clone Detection Strategy. International Wireless Communications and Mobile Computing (IWCMC) 2020:1516–1521. https://doi.org/10.1109/IWCMC48107.2020.9148302

Fakhoury, S, Arnaoudova, V, Noiseux, C, Khomh, F, Antoniol, G (2018) Keep it simple: Is deep learning good for linguistic smell detection? 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), 602–611. https://doi.org/10.1109/SANER.2018.8330265

Fang, C, Liu, Z, Shi, Y, Huang, J, Shi, Q (2020) Functional code clone detection with syntax and semantics fusion learning. Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, 516–527. https://doi.org/10.1145/3395363.3397362

Feng, C, Wang, T, Yu, Y, Zhang, Y, Zhang, Y, Wang, H (2020) Sia-RAE: A Siamese Network based on Recursive AutoEncoder for Effective Clone Detection. 2020 27th Asia-Pacific Software Engineering Conference (APSEC), 238–246. https://doi.org/10.1109/APSEC51365.2020.00032

Fontana FA, Mäntylä MV, Zanoni M, Marino A et al (2016) Comparing and experimenting machine learning techniques for code smell detection. Empir Softw Eng 21:1143–1191. https://doi.org/10.1007/s10664-015-9378-4

Fowler M, Beck K, Brant J, Opdyke W, Roberts D, Gamma E (1999) Refactoring: improving the design of existing code (1 edn). Addison-Wesley Professional

Gao Y, Wang Z, Liu S, Yang L, Sang W, Cai Y (2019) TECCD: A Tree Embedding Approach for Code Clone Detection. IEEE International Conference on Software Maintenance and Evolution (ICSME) 2019:145–156. https://doi.org/10.1109/ICSME.2019.00025

Gentleman, R, Carey, VJ (2008) Unsupervised Machine Learning. In F. Hahne, W. Huber, R. Gentleman, & S. Falcon (Eds.), Bioconductor Case Studies (pp. 137–157). Springer. https://doi.org/10.1007/978-0-387-77240-0_10

Goodfellow I, Bengio Y, Courville A (2016) Deep Learning. MIT Press

Guggulothu T, Moiz SA (2020) Code smell detection using multi-label classification approach. Software Qual J 28(3):1063–1086. https://doi.org/10.1007/s11219-020-09498-y

Guo, X, Shi, C, Jiang, H (2019) Deep semantic-Based Feature Envy Identification. Proceedings of the 11th Asia-Pacific Symposium on Internetware, 1–6. https://doi.org/10.1145/3361242.3361257

Guo C, Yang H, Huang D, Zhang J, Dong N, Xu J, Zhu J (2020) Review Sharing via Deep Semi-Supervised Code Clone Detection. IEEE Access 8:24948–24965. https://doi.org/10.1109/ACCESS.2020.2966532

Hadj-Kacem, M, Bouassida, N (2018) A Hybrid Approach To Detect Code Smells using Deep Learning. Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering, 137–146. https://doi.org/10.5220/0006709801370146

Hadj-Kacem, M, Bouassida, N (2019a) Improving the Identification of Code Smells by Combining Structural and Semantic Information. In T. Gedeon, K. W. Wong, & M. Lee (Eds.), Neural Information Processing (pp. 296–304). Springer International Publishing. https://doi.org/10.1007/978-3-030-36808-1_32

Hadj-Kacem M, Bouassida N (2019b) Deep Representation Learning for Code Smells Detection using Variational Auto-Encoder. International Joint Conference on Neural Networks (IJCNN) 2019:1–8. https://doi.org/10.1109/IJCNN.2019.8851854

Hall T, Beecham S, Bowes D, Gray D, Counsell S (2012) A Systematic Literature Review on Fault Prediction Performance in Software Engineering. IEEE Trans Software Eng 38(6):1276–1304. https://doi.org/10.1109/TSE.2011.103

Hamdy A, Tazy M (2020) Deep Hybrid Features for Code Smells Detection. J Theor Appl Inf Technol 98:2684–2696

He H, Garcia EA (2009) Learning from Imbalanced Data. IEEE Trans Knowl Data Eng 21(9):1263–1284. https://doi.org/10.1109/TKDE.2008.239

Hosseini S, Turhan B, Gunarathna D (2019) A Systematic Literature Review and Meta-Analysis on Cross Project Defect Prediction. IEEE Trans Software Eng 45(2):111–147. https://doi.org/10.1109/TSE.2017.2770124

Hua W, Sui Y, Wan Y, Liu G, Xu G (2021) FCCA: Hybrid Code Representation for Functional Clone Detection Using Attention Networks. IEEE Trans Reliab 70(1):304–318. https://doi.org/10.1109/TR.2020.3001918

Jaiswal A, Babu AR, Zadeh MZ, Banerjee D, Makedon F (2021) A Survey on Contrastive Self-Supervised Learning. Technologies 9(1):2. https://doi.org/10.3390/technologies9010002

Ji X, Liu L, Zhu J (2021) Code Clone Detection with Hierarchical Attentive Graph Embedding. Int J Software Eng Knowl Eng 31(06):837–861. https://doi.org/10.1142/S021819402150025X

Jiang, L, Misherghi, G, Su, Z, Glondu, S (2007) DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. 29th International Conference on Software Engineering (ICSE'07), 96–105. https://doi.org/10.1109/ICSE.2007.30

Jo Y-B, Lee J, Yoo C-J (2021) Two-Pass Technique for Clone Detection and Type Classification Using Tree-Based Convolution Neural Network. Appl Sci 11(14):6613. https://doi.org/10.3390/app11146613

Karabulut EM, Özel SA, İbrikçi T (2012) A comparative study on the effect of feature selection on classification accuracy. Procedia Technol 1:323–327. https://doi.org/10.1016/j.protcy.2012.02.068

Kaur, A, Jain, S, Goel, S, Dhiman, G (2020) A Review on Machine-learning Based Code Smell Detection Techniques in Object-oriented Software System(s). https://doi.org/10.2174/2352096513999200922125839

Kaur A, Saini M (2021) Enhancing the Software Clone Detection in BigCloneBench: A Neural Network Approach. International Journal of Open Source Software and Processes (IJOSSP) 12(3):17–31. https://doi.org/10.4018/IJOSSP.2021070102

Khan MA, Le H, Do K, Tran T, Ghose A, Dam K, Sindhgatta R (2018) Memory-augmented neural networks for predictive process analytics. arXiv preprint. https://arxiv.org/abs/1802.00938. Accessed 07-01-2022

Kim DK (2019) Enhancing code clone detection using control flow graphs. Int J Electric Comput Eng (IJECE) 9(5):3804. https://doi.org/10.11591/ijece.v9i5.pp3804-3812

Kim DK (2020) A Deep Neural Network-Based Approach to Finding Similar Code Segments. IEICE Trans Inf Syst E103D(4):874–878. https://doi.org/10.1587/transinf.2019EDL8195

Kitchenham B (2004) Procedures for performing systematic reviews. Keele, UK, Keele University, 33(2004), 1–26.

Kitchenham B, Charters S (2007) Guidelines for performing systematic literature reviews in software engineering. Technical Report EBSE 2007-001, Keele University and Durham University Joint Report.

Kitchenham B, Pearl Brereton O, Budgen D, Turner M, Bailey J, Linkman S (2009) Systematic literature reviews in software engineering – A systematic literature review. Inf Softw Technol 51(1):7–15. https://doi.org/10.1016/j.infsof.2008.09.009

Kotsiantis SB (2007) Supervised machine learning: a review of classification techniques. Informatica 31:249–268

Lacerda G, Petrillo F, Pimenta M, Guéhéneuc YG (2020) Code smells and refactoring: A tertiary systematic review of challenges and observations. J Syst Softw 167:110610. https://doi.org/10.1016/j.jss.2020.110610

Le QV, Ngiam J, Coates A, Lahiri A, Prochnow B, Ng AY (2011) On optimization methods for deep learning. In Proceedings of the 28th International Conference on International Conference on Machine Learning (ICML'11). Omnipress, Madison, WI, USA, pp 265–272

Lei, M, Li, H, Li, J, Aundhkar, N, Kim, D-K (2022) Deep learning application on code clone detection: A review of current knowledge. J Syst Softw, 184(C). https://doi.org/10.1016/j.jss.2021.111141

Lewowski, T, Madeyski, L (2022) Code Smells Detection Using Artificial Intelligence Techniques: A Business-Driven Systematic Review. In N. Kryvinska & A. Poniszewska-Marańda (Eds.), Developments in Information & Knowledge Management for Business Applications: Volume 3 (pp. 285–319). Springer International Publishing. https://doi.org/10.1007/978-3-030-77916-0_12

Li L, Feng H, Zhuang W, Meng N, Ryder B (2017a) CCLearner: A Deep Learning-Based Clone Detection Approach. IEEE International Conference on Software Maintenance and Evolution (ICSME) 2017:249–260. https://doi.org/10.1109/ICSME.2017.46

Li, Y, Tarlow, D, Brockschmidt, M, Zemel, R (2017b) Gated Graph Sequence Neural Networks (arXiv:1511.05493). arXiv. https://doi.org/10.48550/arXiv.1511.05493

Li, B, Ye, C, Guan, S, Zhou, H (2020a) Semantic Code Clone Detection Via Event Embedding Tree and GAT Network. 2020a IEEE 20th International Conference on Software Quality, Reliability and Security (QRS), 382–393. https://doi.org/10.1109/QRS51102.2020.00057

Li G, Tang Y, Zhang X, Yi B (2020b) A Deep Learning Based Approach to Detect Code Clones. International Conference on Intelligent Computing and Human-Computer Interaction (ICHCI) 2020:337–340. https://doi.org/10.1109/ICHCI51889.2020.00078

Liang H, Ai L (2021) AST-path Based Compare-Aggregate Network for Code Clone Detection. International Joint Conference on Neural Networks (IJCNN) 2021:1–8. https://doi.org/10.1109/IJCNN52387.2021.9534099

Lim T-S, Loh W-Y, Shih Y-S (2000) A Comparison of Prediction Accuracy, Complexity, and Training Time of Thirty-Three Old and New Classification Algorithms. Mach Learn 40(3):203–228. https://doi.org/10.1023/A:1007608224229

Liu, H, Jin, J, Xu, Z, Bu, Y, Zou, Y, Zhang, L (2019) Deep Learning Based Code Smell Detection. IEEE Trans Soft Eng, 1–1. https://doi.org/10.1109/TSE.2019.2936376

Liu, X, Zhang, F, Hou, Z, Wang, Z, Mian, L, Zhang, J, Tang, J (2021) Self-supervised Learning: Generative or Contrastive. ArXiv:2006.08218 [Cs, Stat]. http://arxiv.org/abs/2006.08218

Ma Y, He H (eds) (2013) Imbalanced learning: foundations, algorithms, and applications (1st edn). Wiley-IEEE Press.

Marinescu C, Marinescu R, Mihancea PF, Ratiu D, Wettel R (2005) Iplasma: an integrated platform for quality assessment of object-oriented design. ICSM, pp 77–80

Mayvan BB, Rasoolzadegan A, Jafari AJ (2020) Bad smell detection using quality metrics and refactoring opportunities. J Softw Evol Process 32(8):e2255. https://doi.org/10.1002/smr.2255

Mehrotra, N, Agarwal, N, Gupta, P, Anand, S, Lo, D, Purandare, R (2021) Modeling Functional Similarity in Source Code with Graph-Based Siamese Networks. IEEE Trans Softw Eng, 1–1. https://doi.org/10.1109/TSE.2021.3105556

Meng Y, Liu L (2020) A Deep Learning Approach for a Source Code Detection Model Using Self-Attention. Complexity 2020:1–15. https://doi.org/10.1155/2020/5027198

Menshawy, RS, Yousef, AH, Salem, A (2021) Code Smells and Detection Techniques: A Survey. 2021 International Mobile, Intelligent, and Ubiquitous Computing Conference (MIUCC), 78–83. https://doi.org/10.1109/MIUCC52538.2021.9447669

Moha N, Gueheneuc Y-G, Duchien L, Le Meur A-F (2010) DECOR: A Method for the Specification and Detection of Code and Design Smells. IEEE Trans Softw Eng 36(1):20–36. https://doi.org/10.1109/TSE.2009.50

Mostaeen G, Roy B, Roy CK, Schneider K, Svajlenko J (2020) A machine learning based framework for code clone validation. J Syst Softw 169:110686. https://doi.org/10.1016/j.jss.2020.110686

Mumtaz H, Alshayeb M, Mahmood S, Niazi M (2019) A survey on UML model smells detection techniques for software refactoring. J Softw Evol Process 31(3):e2154. https://doi.org/10.1002/smr.2154

Nafi KW, Roy B, Roy CK, Schneider KA (2020) A universal cross language software similarity detector for open source software categorization. J Syst Softw 162:110491. https://doi.org/10.1016/j.jss.2019.110491

Nafi, KW, Kar, TS, Roy, B, Roy, CK, Schneider, KA (2019) CLCDSA: Cross Language Code Clone Detection using Syntactical Features and API Documentation. 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), 1026–1037. https://doi.org/10.1109/ASE.2019.00099

Nair, A, Roy, A, Meinke, K (2020) funcGNN: A Graph Neural Network Approach to Program Similarity. Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 1–11. https://doi.org/10.1145/3382494.3410675

Ohri K, Kumar M (2021) Review on self-supervised image recognition using deep neural networks. Knowl Based Syst 224:107090. https://doi.org/10.1016/j.knosys.2021.107090

Olbrich SM, Cruzes DS, Sjøberg DIK (2010) Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems. IEEE International Conference on Software Maintenance 2010:1–10. https://doi.org/10.1109/ICSM.2010.5609564

Palomba, F, Di Nucci, D, Tufano, M, Bavota, G, Oliveto, R, Poshyvanyk, D, De Lucia, A (2015) Landfill: An Open Dataset of Code Smells with Public Evaluation. 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, 482–485. https://doi.org/10.1109/MSR.2015.69

Patnaik A, Padhy N (2021) A Hybrid Approach to Identify Code Smell Using Machine Learning Algorithms. International Journal of Open Source Software and Processes 12(2):21–35. https://doi.org/10.4018/IJOSSP.2021040102

Pecorelli F, Nucci DD, Roover CD, Lucia AD (2020) A large empirical assessment of the role of data balancing in machine-learning-based code smell detection. J Syst Softw. https://doi.org/10.1016/j.jss.2020.110693

Perez, D, Chiba, S (2019) Cross-Language Clone Detection by Learning Over Abstract Syntax Trees. 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), 518–528. https://doi.org/10.1109/MSR.2019.00078

Pérez, J (2013) Refactoring Planning for Design Smell Correction: Summary, Opportunities and Lessons Learned. Proceedings of the 2013 IEEE International Conference on Software Maintenance, 572–577. https://doi.org/10.1109/ICSM.2013.98

Rasmussen CE, Ghahramani Z (2001) Occam's Razor. In Advances in Neural Information Processing Systems 13:294–300

Ren, S, Shi, C, Zhao, S (2021) Exploiting Multi-aspect Interactions for God Class Detection with Dataset Fine-tuning. 2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC), 864–873. https://doi.org/10.1109/COMPSAC51774.2021.00119

Sabir F, Palma F, Rasool G, Guéhéneuc Y-G, Moha N (2019) A systematic literature review on the detection of smells and their evolution in object-oriented and service-oriented systems. Softw Practice Experience 49(1):3–39. https://doi.org/10.1002/spe.2639

Saini, V, Farmahinifarahani, F, Lu, Y, Baldi, P, Lopes, CV (2018) Oreo: Detection of clones in the twilight zone. Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 354–365. https://doi.org/10.1145/3236024.3236026

Saini, V, Farmahinifarahani, F, Lu, Y, Yang, D, Martins, P, Sajnani, H, Baldi, P, Lopes, CV (2019) Towards Automating Precision Studies of Clone Detectors. 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), 49–59. https://doi.org/10.1109/ICSE.2019.00023

Sajnani, H, Saini, V, Svajlenko, J, Roy, C K, Lopes, CV (2016) SourcererCC: Scaling Code Clone Detection to Big-Code. 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), 1157–1168. https://doi.org/10.1145/2884781.2884877

Sammut C, Webb GI (2011) Encyclopedia of machine learning. Springer Sci Bus Med. https://doi.org/10.1007/978-0-387-30164-8

Sarker IH (2021) Deep Learning: A Comprehensive Overview on Techniques, Taxonomy, Applications and Research Directions. SN Computer Science 2(6):420. https://doi.org/10.1007/s42979-021-00815-1

Sharma T, Efstathiou V, Louridas P, Spinellis D (2021) Code smell detection by deep direct-learning and transfer-learning. J Syst Softw 176:110936. https://doi.org/10.1016/j.jss.2021.110936

Sheneamer A, Roy S, Kalita J (2021) An Effective Semantic Code Clone Detection Framework Using Pairwise Feature Fusion. IEEE Access 9:84828–84844. https://doi.org/10.1109/ACCESS.2021.3079156

Sheneamer, A, Hazazi, H, Roy, S, Kalita, J (2017) Schemes for Labeling Semantic Code Clones using Machine Learning. 2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA), 981–985. https://doi.org/10.1109/ICMLA.2017.00-25

Sheneamer, A (2018) CCDLC Detection Framework-Combining Clustering with Deep Learning Classification for Semantic Clones. 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA), 701–706. https://doi.org/10.1109/ICMLA.2018.00111

Sidhu, BK, Singh, K, Sharma, N (2020) A machine learning approach to software model refactoring. Int J Comput Appl, 1–12. https://doi.org/10.1080/1206212X.2020.1711616

Storey, M-A, Zagalsky, A (2016) Disrupting developer productivity one bot at a time. Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 928–931. https://doi.org/10.1145/2950290.2983989

Suryanarayana, G, Samarthyam, G, Sharma, T (2015) Refactoring for Software Design Smells: Managing Technical Debt, Chapter 2—Design Smells. In G. Suryanarayana, G. Samarthyam, & T. Sharma (Eds.), Refactoring for Software Design Smells (pp. 9–19). Morgan Kaufmann. https://doi.org/10.1016/B978-0-12-801397-7.00002-3

Sutskever I, Martens J, Dahl G, Hinton G (2013) On the importance of initialization and momentum in deep learning. Proceedings of the 30th International Conference on Machine Learning, 1139–1147. https://proceedings.mlr.press/v28/sutskever13.html. Accessed 01 Jan 2022

Svajlenko J, Islam JF, Keivanloo I, Roy CK, Mia MM (2014) Towards a Big Data Curated Benchmark of Inter-project Code Clones. IEEE International Conference on Software Maintenance and Evolution 2014:476–480. https://doi.org/10.1109/ICSME.2014.77

Tantithamthavorn C, McIntosh S, Hassan AE, Matsumoto K (2019) The Impact of Automated Parameter Optimization on Defect Prediction Models. IEEE Trans Software Eng 45(7):683–711. https://doi.org/10.1109/TSE.2018.2794977

Tsantalis, N, Chaikalis, T, Chatzigeorgiou, A (2008) JDeodorant: Identification and Removal of Type-Checking Bad Smells. 2008 12th European Conference on Software Maintenance and Reengineering, 329–331. https://doi.org/10.1109/CSMR.2008.4493342

Tufano, M, Watson, C, Bavota, G, Di Penta, M, White, M, Poshyvanyk, D (2018) Deep learning similarities from different representations of source code. Proceedings of the 15th International Conference on Mining Software Repositories, 542–553. https://doi.org/10.1145/3196398.3196431

Ullah F, Naeem MR, Mostarda L, Shah SA (2021) Clone detection in 5G-enabled social IoT system using graph semantics and deep learning model. Int J Mach Learn Cybern 12(11):3115–3127. https://doi.org/10.1007/s13042-020-01246-9

Wang W, Li G, Shen S, Xia X, Jin Z (2020c) Modular Tree Network for Source Code Representation Learning. ACM Transactions on Software Engineering and Methodology 29(4):1–23. https://doi.org/10.1145/3409331

Wang, C, Gao, J, Jiang, Y, Xing, Z, Zhang, H, Yin, W, Gu, M, Sun, J (2019) Go-clone: Graph-embedding based clone detector for Golang. Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, 374–377. https://doi.org/10.1145/3293882.3338996

Wang, H, Liu, J, Kang, J, Yin, W, Sun, H, Wang, H (2020a). Feature Envy Detection based on Bi-LSTM with Self-Attention Mechanism. 2020a IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom), 448–457. https://doi.org/10.1109/ISPA-BDCloud-SocialCom-SustainCom51426.2020.00082

Wang, W, Li, G, Ma, B, Xia, X, Jin, Z (2020b) Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree. 2020b IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), 261–271. https://doi.org/10.1109/SANER48275.2020.9054857

Wei, H, Li, M (2017) Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code. Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, 3034–3040. https://doi.org/10.24963/ijcai.2017/423

Wei, H-H, Li, M (2018) Positive and Unlabeled Learning for Detecting Software Functional Clones with Adversarial Training. Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, 2840–2846. https://doi.org/10.24963/ijcai.2018/394

Wen J, Li S, Lin Z, Hu Y, Huang C (2012) Systematic literature review of machine learning based software development effort estimation models. Inf Softw Technol 54(1):41–59. https://doi.org/10.1016/j.infsof.2011.09.002

White M, Tufano M, Vendome C, Poshyvanyk D (2016) Deep learning code fragments for code clone detection. 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), 87–98. https://doi.org/10.1145/2970276.2970326

Wu, Y, Zou, D, Dou, S, Yang, S, Yang, W, Cheng, F, Liang, H, Jin, H (2020) SCDetector: Software functional clone detection based on semantic tokens analysis. Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, 821–833. https://doi.org/10.1145/3324884.3416562

Wu Y, Wang W (2021) Code Similarity Detection Based on Siamese Network. IEEE International Conference on Information Communication and Software Engineering (ICICSE) 2021:47–51. https://doi.org/10.1109/ICICSE52190.2021.9404110

Xie C, Wang X, Qian C, Wang M (2020) A Source Code Similarity Based on Siamese Neural Network. Appl Sci 10(21):7519. https://doi.org/10.3390/app10217519

Xu, W (2021) Multi-Granularity Code Smell Detection using Deep Learning Method based on Abstract Syntax Tree. 503–509. https://doi.org/10.18293/SEKE2021-014

Xue, H, Venkataramani, G, Lan, T (2018) Clone-Slicer: Detecting Domain Specific Binary Code Clones through Program Slicing. Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation - FEAST '18, 27–33. https://doi.org/10.1145/3273045.3273047

Yamashita A, Counsell S (2013) Code smells as system-level indicators of maintainability: An empirical study. J Syst Softw 10(86):2639–2653. https://doi.org/10.1016/j.jss.2013.05.007

Yin, X, Shi, C, Zhao, S (2021) Local and Global Feature Based Explainable Feature Envy Detection. 2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC), 942–951. https://doi.org/10.1109/COMPSAC51774.2021.00127

Yu, H, Lam, W, Chen, L, Li, G, Xie, T, Wang, Q (2019) Neural Detection of Semantic Code Clones Via Tree-Based Convolution. 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC), 70–80. https://doi.org/10.1109/ICPC.2019.00021

Yuan, Y, Kong, W, Hou, G, Hu, Y, Watanabe, M, Fukuda, A (2020) From Local to Global Semantic Clone Detection. 2019 6th International Conference on Dependable Systems and Their Applications (DSA), 13–24. https://doi.org/10.1109/DSA.2019.00012

Zeng J, Ben K, Li X, Zhang X (2019) Fast Code Clone Detection Based on Weighted Recursive Autoencoders. IEEE Access 7:125062–125078. https://doi.org/10.1109/ACCESS.2019.2938825

Zhang Y, Wang T (2021) CCEyes: An Effective Tool for Code Clone Detection on Large-Scale Open Source Repositories. IEEE International Conference on Information Communication and Software Engineering (ICICSE) 2021:61–70. https://doi.org/10.1109/ICICSE52190.2021.9404141

Zhang M, Hall T, Baddoo N (2011) Code Bad Smells: A review of current knowledge. J Softw Maint Evol Res Pract 23(3):179–202. https://doi.org/10.1002/smr.521

Zhang L, Feng Z, Ren W, Luo H (2020) Siamese-Based BiLSTM Network for Scratch Source Code Similarity Measuring. International Wireless Communications and Mobile Computing (IWCMC) 2020:1800–1805. https://doi.org/10.1109/IWCMC48107.2020.9148382

Zhang, J, Wang, X, Zhang, H, Sun, H, Wang, K, Liu, X (2019) A Novel Neural Source Code Representation Based on Abstract Syntax Tree. 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), 783–794. https://doi.org/10.1109/ICSE.2019.00086

Zhang, J, Hong, H, Zhang, Y, Wan, Y, Liu, Y, Sui, Y (2021) Disentangled Code Representation Learning for Multiple Programming Languages. Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021, 4454–4466. https://doi.org/10.18653/v1/2021.findings-acl.391

Zhao, G, Huang, J (2018) DeepSim: Deep learning code functional similarity. Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 141–151. https://doi.org/10.1145/3236024.3236068

Zhou, X, Jin, Y, Zhang, H, Li, S, Huang, X (2016) A Map of Threats to Validity of Systematic Literature Reviews in Software Engineering. 2016 23rd Asia-Pacific Software Engineering Conference (APSEC), 153–160. https://doi.org/10.1109/APSEC.2016.031