Knowledge-Based Systems 255 (2022) 109737

Contents lists available at ScienceDirect

Knowledge-Based Systems

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

DeleSmell: Code smell detection based on deep learning and latent semantic analysis

Yang Zhang [a,*], Chuyan Ge [a], Shuai Hong [a], Ruili Tian [b], Chunhao Dong [c], Jingjing Liu [a]

[a] School of Information Science and Engineering, Hebei University of Science and Technology, Shijiazhuang, Hebei, 050018, China
[b] China Unicom Corporation, Shijiazhuang, Hebei, 050000, China
[c] School of Computer, Beijing Institute of Technology, Beijing, 100081, China

ARTICLE INFO

Article history:
Received 17 April 2022
Received in revised form 16 August 2022
Accepted 16 August 2022
Available online 22 August 2022

Keywords:
Code smell
Deep learning
Refactoring
Latent semantic analysis

ABSTRACT

The presence of code smells will increase the risk of failure, make software difficult to maintain, and introduce potential technique debt in the future. Although many deep-learning-based approaches have been proposed to detect code smells, most existing works suffer from the problem of incomplete feature extraction and unbalanced distribution between positive samples and negative samples. Furthermore, the accuracy of existing works can be further improved. This paper proposes a novel approach named *DeleSmell* to detect code smells based on a deep learning model. The dataset is built by extracting samples from 24 real-world projects. To improve the imbalance in the dataset, a refactoring tool is developed to automatically transform good source code into smelly code and to generate positive samples based on real cases. *DeleSmell* collects both structural features through *iPlasma* and semantic features via latent semantic analysis and *word2vec*. DeleSmell's model includes a convolutional neural network(CNN) branch and gate recurrent unit(GRU)-attention branch. The final classification is conducted by an support vector machine(SVM). In the experimentation, the effectiveness of *DeleSmell* is evaluated by answering seven research questions. The experimental results show that *DeleSmell* improves the accuracy of brain class (BC) and brain method (BM) code smells detection by up to 4.41% compared with existing approaches, demonstrating the effectiveness of our approach.

1. Introduction

Code smells are not bugs of a program but poor implementation choices that may signify the weakness of software design. Although it is a surface indication of code structure, code smells may expose a deeper problem in a software system. The presence of code smells may increase the risk of failure, make software difficult to maintain, and introduce potential technique debt in the future.

Detecting code smells has become increasingly important for software evolution. More than 20 kinds of code smells have been identified in thousands of real-world projects. Palomba et al. [1] conducted a survey with 162 developers and discovered that more than 90% of developers focus on prevalent code smells (such as feature envy, god class, and long method) rather than seldom-used code smells, such as brain class (BC) and brain method (BM). BC/BM code smells usually have numerous structural branches and deeply recursive invocations, which disobeys the proportion rule and increases the risk of failure.

Many works have been conducted to identify code smells since the automated detection tool was originally proposed [2]. Traditional approaches employ heuristic-based approaches [3,4] by computing a collection of metrics. A predefined threshold is set to distinguish between smelly samples and nonsmelly samples. However, code snippets with different behaviors may use the same metric and consequently have a totally different code smells. Furthermore, choosing a proper threshold requires expertise. Lacking experience and holding potential bias have a negative impact on making the appropriate decision. Some researchers have improved code smells detection by employing machine learning (ML) techniques, such as K-means [5], support vector machine (SVM) [6], and naive bayes [7]. However, these techniques suffer from insufficient feature extraction [8]. As a result, the accuracy of their results is not satisfactory. In recent years, some researchers have begun to detect code smells by leveraging deep learning techniques, including convolutional neural network(CNN) [9], long short-term memory(LSTM) [10], and residual network(ResNet) [11]. Most works initially build a dataset by extracting features from the source code and then train a deep learning model to obtain a classifier. Deep-learning-based approaches significantly improve the accuracy of code smell detection.

* Corresponding author.
E-mail address: zhangyang@hebust.edu.cn (Y. Zhang).

https://doi.org/10.1016/j.knosys.2022.109737
0950-7051/

Although many works on code smell detection have been conducted, several problems remain open.

First, existing approaches mainly focus on structural features of the source code, while semantic features are prone to being disregarded. Consequently, the features learned by the model may be biased. Second, existing works lack a unified dataset for different code smells. Additionally, the deep-learning-based model can be further optimized. Third, the existing data enhancement algorithm synthetic minority oversampling technique (SMOTE) [12] is not visualized. Some samples generated by SMOTE probably disturb the boundary between positive samples and negative samples (illustrated in Section 3).

To address these problems, this paper proposes a novel approach named *DeleSmell* to detect code smells based on a deep learning model. To increase the number of positive samples, a refactoring tool is developed to automatically transform nonsmelly items into smelly items in real-world projects. *DeleSmell* extracts both structural features by iPlasma [13] and semantic features by latent semantic analysis (LSA) [14] and *word2vec* [15]. Thus, a dataset with more than 200,000 samples is constructed from 24 real-world projects. *DeleSmell* builds a deep neural network model, including a CNN branch and gate recurrent unit(GRU)-attention branch. The final classification is conducted by an SVM. In the experimentation, the effectiveness of *DeleSmell* is evaluated by answering seven research questions. The experimental results show that *DeleSmell* improves the accuracy of BC/BM code smells detection by up to 4.41% compared with existing approaches, demonstrating the effectiveness of our approach.

The main contributions of this paper can be summarized as follows:

- A novel approach named *DeleSmell* is proposed to detect code smells based on the deep learning model and LSA.
- A refactoring tool is developed to automatically generate positive samples of the dataset.
- A dataset with more than 200,000 samples is built from 24 real-world projects.
- Many experiments have been conducted to evaluate the effectiveness of *DeleSmell* by answering seven research questions.

The remainder of the paper is structured as follows: Section 2 examines related works. Section 3 illustrates the motivation of our work. Section 4 presents an overview of *DeleSmell* and the design of each component. The experimental setup and results are reported in Section 5. This paper is concluded in Section 6.

2. Related works

In recent years, many works on code smell detection by heuristic-based, ML-based, and deep-learning-based approaches have been conducted [16–20]. In this section, we present the background on code smell detection and elaborate on the related literature.

2.1. Heuristic based detection

The heuristic-based approach identifies the symptoms of a code smell and then chooses a set of thresholds for some metrics to detect code smells. Tsantalis et al. [21] presented a tool *JDeodorant* to detect code smells. The empirical evaluation of the performance of this tool demonstrates an average improvement in accuracy of 75%. Marinescu et al. [13] proposed a detection tool *iPlasma* to detect code problems such as identity disharmonies, collaboration disharmonies and classification disharmonies. *iPlasma* is an integrated platform for evaluating the quality of object-oriented software from several aspects, such as model extraction and high-level metric analysis. The detection tool *DECOR* [3] defined a set of approaches for automatic design and code smell detection (also known as Anti-Pattern). *DECOR* combines metrics with thresholds and extracts the features affected by code smell. This detection tool identifies code smell and automatically generates the detection algorithms by templates. However, since different tools have different rules for detecting code smell, it is possible that different results can be obtained for the same code.

2.2. Machine-learning-based detection

Some researchers have leveraged machine-learning-based techniques for code smell detection. Machine learning techniques utilize a supervised learning approach to analyze a sufficiently large number of samples and train them. In contrast to heuristics, this approach relies on classifiers to distinguish whether the source code is smelly, rather than defining thresholds based on computational metrics. Amorim et al. [22] presented an empirical report that evaluates the use of a decision tree classification algorithm to identify code smells in different projects. Fontana et al. [23] applied several machine learning models and compared the performance of these models by considering four performance metrics. Guggulothu et al. [5] employed multilabel classification to detect whether a given code element is affected by multiple code smells. Alazba et al. [24] proposed an integrated learning approach to detect code smells. The output of an ML classifier is combined and then re-output. The results are predicted after fitting the output of all the basic classifiers. The experimental results show that the accuracy of integrated learning is higher than that of a single ML classifier.

2.3. Deep-learning-based detection

To learn the deeper features, researchers have started to use deep learning models for code smell detection. Ananta et al. [9] employed a one-dimensional convolutional neural network to detect BC/BM and obtained 97% accuracy and 94% accuracy, respectively. Palomba et al. [1] proposed a code smell intensity prediction model that relies on both technical aspects and community-related aspects. Liu et al. [25] proposed an automatic approach to generating labeled training data for the neural network-based classifier, which does not require any human intervention. Sharma et al. [10] applied deep learning models in detecting code smells without carrying out extensive feature engineering and compared the performance of different models. Their exploration not only shows the feasibility of applying transfer learning for identifying code smells but also compares the performance of deep learning models in the transfer learning context.

3. Motivation

This section presents the motivation to demonstrate the rationale. Building the dataset is important for deep-learning-based detection. During this procedure, it is important to maintain the balance between positive samples and negative samples. Pecorelli et al. [26] empirically compared five techniques of code smell detection and suggested that SMOTE obtained the best performance. MARS [11] is a detection approach for BC/BM code smells and balances the dataset by leveraging SMOTE. However, this data enhancement approach suffers from several problems. SMOTE randomly selects minorities as candidates to automatically generate similar samples. Not all samples are suitable to be selected for expansion. According to the principle of SMOTE, we illustrate a

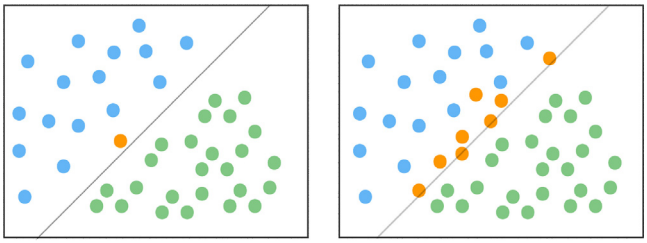


Fig. 1. Data enhancement of SMOTE.

possible situation in Fig. 1. The blue dots and green dots represent positive samples and negative samples, respectively. The orange dots located near the boundary represent the seeds chosen by SMOTE. The new samples generated by SMOTE probably become closer to the boundary, as shown in Fig. 1(b). As a result, the boundary between positive samples and negative samples may become disturbed, resulting in incorrect classification results.

Existing approaches [11,27] rely on structural features of source codes, while semantic features are prone to being disregarded. The difference between semantic analysis and structural analysis lies in the procedure of feature extraction. The semantic features leverage text mining algorithms to calculate the text similarity. Palomba et al. [27] utilized structural and semantic analysis to obtain version information of several Java projects. The results highlight that code smell detected through semantic analysis are generally perceived as dangerous as code smells detected through structural features, even if they do not exceed any structural metrics' thresholds. Additionally, the results emphasize that the structural and semantic information are complementary; however, few works have been conducted to combine this information. Furthermore, most semantic analysis approaches only consider identifiers and comments and do not analyze semantic features of the source code. This approach is limited if the identifier or method is not named according to its function or if the comment is not relevant to the method.

4. Design

An overview of *DeleSmell* is depicted in Fig. 2. A total of 24 real-world projects are selected to generate the dataset. To avoid the imbalance between positive samples and negative samples, we developed a refactoring tool to automatically generate positive samples for BC/BM code smells. Both semantic and structural features are extracted from these projects. To obtain the semantic features of the source code, we generate the word vectors by *word2vec* and calculate the average conceptual similarity (ACSM) by LSA. Structural features are extracted by employing code metrics such as coupling between two object classes (CBO), access to foreign data (ATFD), lines of code (LOC), etc., based on iPlasma [13]. The dataset is built by extracting samples from 24 real-world projects. DeleSmell's model includes a CNN branch and a GRU-attention branch. The final classification is conducted by an SVM. The output indicates whether the code entity is smelly.

4.1. Data collection

We generate the dataset by taking a two-pronged approach for data collection. First, we conduct a breadth-oriented study by collecting 24 real-world projects from *GitHub*.[1] Seventeen of 24 projects include more than 500 classes (NOC), while 15 of 24 projects contain more than 5,000 number of methods (NOM). Furthermore, the LOC of most projects is substantially greater than 30,000. Table 1 presents projects and their configurations. Second, we conduct a depth-oriented study on the features of the source code by semantic and structural analysis, which will be illustrated in Section 4.3.

[1] https://github.com/

4.2. Automatic items generation

To prevent underfitting, we need numerous samples to improve the generalization of the neural network. However, the number of code smells in existing open-source projects is not enough to support the training of neural networks, which results in an imbalanced the dataset. To solve this problem, we generate smelly code by refactoring. Different from existing refactoring applied to smelly software to improve software quality, our refactoring is applied to well-designed programs to generate code smells.

We develop a refactoring tool to convert a normal method into a BM. This method can generate numerous labeled training samples for deep-learning-based detection. A method is considered to be a BM if an increasing number of functions are added to this method until it becomes out of control [28]. If we merge multiple methods into one method, the integrated method has multiple roles and thus can be considered a BM. *DeleSmell* employs the rules [9] to label the items. This tool automatically generates items without any human intervention. Thus, we can obtain a large number of labeled items from all kinds of projects.

An overview of the refactoring framework is shown in Fig. 3. The tool begins by automatically abstracting the source code into an abstract syntax tree (AST) and by generating a call graph. The tool walks through the AST to locate a method m that could be a positive item through refactoring. m is selected by the following rules: (a) the cyclomatic complexity (CYCLO) of m satisfies Eq. (1), where C_m represents the CYCLO of method m and N represents the number of methods in a project. (b) m is not polymorphic. If several methods overwrite in children class m, it cannot be inlined [29]. The call graph is then analyzed to obtain the invocation of the selected method m. We develop the refactoring tool under the *Eclipse* JDT framework and use these analyses and rules to ensure that the refactoring will correctly generate positive items.

$$C_m \geq \frac{1}{N}\sum_{i=1}^{N}(C_i) \tag{1}$$

An example of our refactoring tool is presented in Fig. 4. This example is selected from class *Trading* of project *JAdventure* [30]. The original source code is shown on the left-hand side, while the source code after refactoring is illustrated on the right-hand side. Method *trade*() originally had no BM code smell. If we replace *playerBuy*() and *playerSell*() with their method bodies, the LOC and complexity of method *trade*() become significantly increased. As a result, method *trade*() is transformed into a smelly method.

4.3. Feature extraction

This section presents how to extract semantic and structural features from the projects.

4.3.1. Semantic features extraction

To extract the semantic features from the source code, LSA [31] and *word2vec* [15] are employed to conduct semantic analysis. We build an AST to locate each method for feature extraction. Unlike the traditional approach [14,32], which only extracts the name of a class/method, we focus on all the statement of the code.

The average concept similarity (ACSM) [14] is calculated to measure the cohesion of the source code by LSA. For each method,

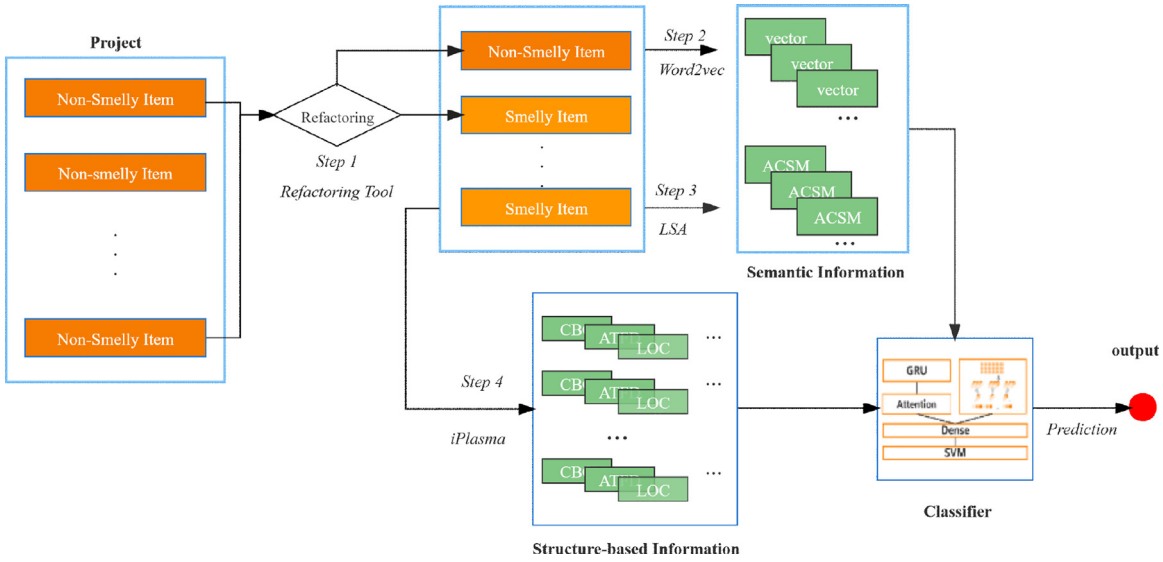


Fig. 2. Overview of *DeleSmell*.

Table 1
Projects and their configurations.

Projects	Description	NOC	NOM	LOC
Anthelion	A Nutch plugin for focused crawling of semantic data	357	2480	32,756
Argouml	A UML based visualization tool	1953	17,456	160,295
Batik	A Java based toolkit for projects	1682	16,247	166,673
Cassandra	Open-source NoSQL database	1004	9770	79,110
Cayenne	Open-source persistence framework	2928	16,997	135,020
Displaytag	Open-source suite of custom tags	262	945	9019
Drjava	A lightweight programming environment for Java	1145	16,920	147,284
Freecol	Turn-based strategy game	344	3066	30,581
Freedomotic	Secure IoT development framework	501	3867	33,857
Freemind	A premier free mind-mapping software written in Java	532	7303	65,866
Gantt	Desktop project scheduling and management tool	685	5848	40,009
Hadoop	Distributed System Infrastructure	1340	13,464	158,560
HSQLDB	Open-source Java database	549	1158	206,860
Itext7	A powerful PDF toolkit	1518	12,771	124,562
JavaStud	Java tutorial example series	218	459	4229
Jedit	Text editor	584	7376	103,503
Jgroups	A Toolkit for Reliable group communication	273	44,470	31,174
Maven	Project management tool	712	4008	23,435
Nutch	A highly extensible Web crawler	366	1983	23,036
Parallelcolt	High-performance scientific computing in Java	1130	14,527	216,685
Pmd	A source code analyzer	2166	9902	50,510
SPECjbb	Java server Business Benchmark	76	746	12,762
Xalan	XSLT processor for transforming XML documents	964	10,359	188,637
Xerces	XML document parsing Open-source project	838	10,683	142,137
Total		**22,127**	**232,805**	**2,186,560**

the frequency of each word is counted to build a method-word matrix, in which each row represents the vector of the method and each column represents the vector of the word. The matrix is compressed by singular value decomposition (SVD) [33]. As a result, the original matrix is transformed into a dense matrix. The similarity between any two methods in latent semantic space [14] can be calculated by Eq. (2).

$$\mathrm{Sim}\left(x_i, x_j\right) = \frac{v_{x_i}^T v_{x_j}}{\left\|v_{x_i}\right\|_2 \times \left\|v_{x_j}\right\|_2} \tag{2}$$

where v_{x_i} represents the vector of the method x_i, $v_{x_i}^T$ represents the transposed matrix of v_{x_i}, and $\left\|v_{x_i}\right\|_2$ denotes the norm of v_{x_i}.

We evaluate the complexity of each class by calculating the ACSM [14] shown in Eq. (3).

$$ACSM = \frac{1}{N} \times \sum_{i=1}^{N} Sim(x_i, x_j) \tag{3}$$

where N represents the number of combinations of any two methods in a specific class.

To demonstrate how to extract semantic information, we select a class with 6 methods (denoted by M1-M6) and 10 words (denoted by W1-W10) as an example. The frequency of each word is counted to build the method-word matrix X shown in Table 2.

The matrix X can be decomposed into three components by SVD. We set k to 3. The decomposed matrix is shown as Eq. (4).

$$\mathrm{X} \approx \mathrm{U_k}\Sigma_\mathrm{k}\mathrm{V_k^T} = \begin{bmatrix} \mathrm{u}_1 & \mathrm{u}_2 & \mathrm{u}_3 \end{bmatrix} \begin{bmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \\ 0 & 0 & \sigma_3 \end{bmatrix} \begin{bmatrix} \mathrm{v}_1^\mathrm{T} \\ \mathrm{v}_2^\mathrm{T} \\ \mathrm{v}_3^\mathrm{T} \end{bmatrix} \tag{4}$$

The dimension of X needs to be reduced to strengthen the code semantic information. As shown in Eq. (5), the first singular values in $\mathrm{U_k}$, Σ_k and $\mathrm{V_k^T}$ are selected to generate U', Σ', and V'

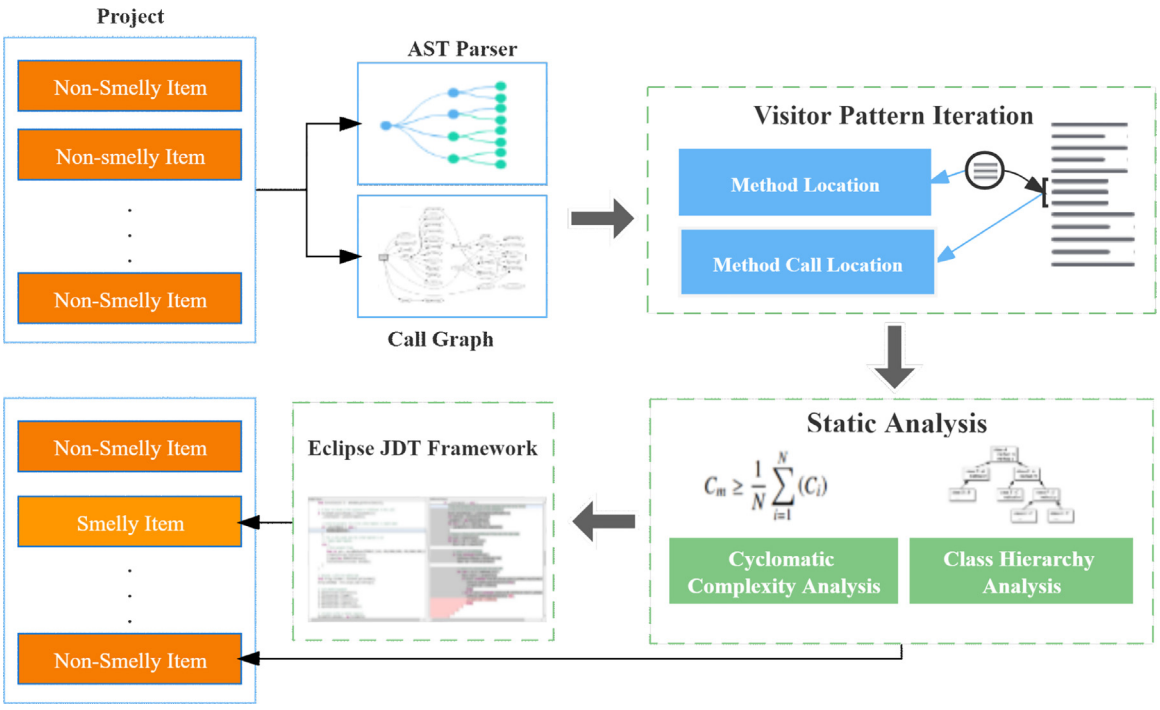
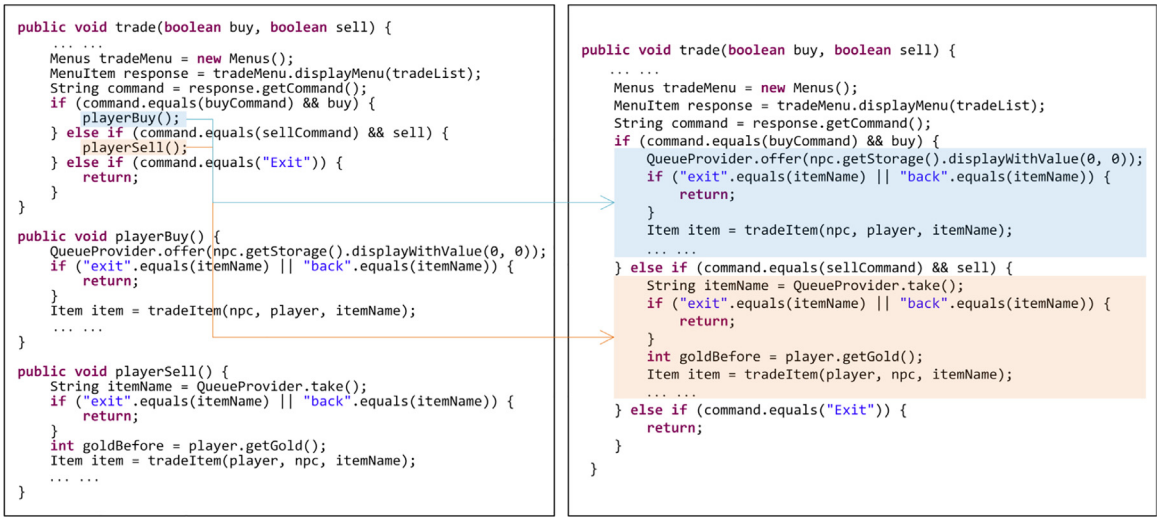


Fig. 3. An overview of refactoring tool.



```
public void trade(boolean buy, boolean sell) {
    ... ...
    Menus tradeMenu = new Menus();
    MenuItem response = tradeMenu.displayMenu(tradeList);
    String command = response.getCommand();
    if (command.equals(buyCommand) && buy) {
        playerBuy();
    } else if (command.equals(sellCommand) && sell) {
        playerSell();
    } else if (command.equals("Exit")) {
        return;
    }
}

public void playerBuy() {
    QueueProvider.offer(npc.getStorage().displayWithValue(0, 0));
    if ("exit".equals(itemName) || "back".equals(itemName)) {
        return;
    }
    Item item = tradeItem(npc, player, itemName);
    ... ...
}

public void playerSell() {
    String itemName = QueueProvider.take();
    if ("exit".equals(itemName) || "back".equals(itemName)) {
        return;
    }
    int goldBefore = player.getGold();
    Item item = tradeItem(player, npc, itemName);
    ... ...
}
```

```
public void trade(boolean buy, boolean sell) {
    ... ...
    Menus tradeMenu = new Menus();
    MenuItem response = tradeMenu.displayMenu(tradeList);
    String command = response.getCommand();
    if (command.equals(buyCommand) && buy) {
        QueueProvider.offer(npc.getStorage().displayWithValue(0, 0));
        if ("exit".equals(itemName) || "back".equals(itemName)) {
            return;
        }
        Item item = tradeItem(npc, player, itemName);
        ... ...
    } else if (command.equals(sellCommand) && sell) {
        String itemName = QueueProvider.take();
        if ("exit".equals(itemName) || "back".equals(itemName)) {
            return;
        }
        int goldBefore = player.getGold();
        Item item = tradeItem(player, npc, itemName);
        ... ...
    } else if (command.equals("Exit")) {
        return;
    }
}
```

Fig. 4. An example of refactoring.

according to the approach in [14].

$$X \approx X' = U' \Sigma' V'^{T} = \begin{bmatrix} u_1 \end{bmatrix} \begin{bmatrix} \sigma_1 \end{bmatrix} \begin{bmatrix} v_1^T \end{bmatrix} \quad (5)$$

The similarity between two methods can be calculated according to matrix X' by Eq. (2). Table 3 shows the similarity between each method in the class.

Eq. (3) is used to calculate the ACSM. It can be concluded that the ACSM of this class is 0.83. If a class or a method has a high ACSM, their function is concentrated, and therefore, the cohesiveness is higher. Otherwise, a low semantic similarity indicates a lower cohesiveness. Furthermore, *word2vec* is leveraged to map methods into vectors. *DeleSmell* converts the source text of a method into a matrix via method *CountVectorizer*() of *sklearn*.[2] The window size is defined as 3 [34], and the one-hot encoding is transformed into 3 dense vectors by the *Skip-Gram* training approach of *word2vec*.

Table 2
The method-word matrix.

	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10
M1	0	1	2	0	0	1	0	0	0	1
M2	0	0	0	1	0	0	1	1	0	0
M3	1	1	1	0	2	0	0	0	0	3
M4	1	1	1	0	1	0	0	1	1	3
M5	0	1	0	0	1	2	0	0	0	0
M6	2	0	1	0	0	0	1	0	0	1

Semantic features can help boost the performance of *DeleSmell*. First, *DeleSmell* extracts the features from the source code, which provides sufficient information to conduct semantic analysis. Different from the existing approaches, *DeleSmell* analyzes

[2] https://scikit-learn.org/

Table 3
Similarity between each method.

	M1	M2	M3	M4	M5	M6
M1	–	0.18	1.38	1.38	0.76	1.17
M2	0.18	–	0.21	0.21	0.12	0.18
M3	1.38	0.21	–	1.61	0.89	1.37
M4	1.38	0.21	1.61	–	0.89	1.36
M5	0.76	0.12	0.89	0.89	–	0.75
M6	1.17	0.18	1.37	1.36	0.75	–

not only the comments and identifiers of the code but also the information of the code statement. Second, LSA is proposed on the basis of mathematical theory and does not need any human intervention. The value of ACSM can reveal cohesiveness, which is an important indicator of brain code smells detection. Therefore, we can effectively identify the code smell. Third, both LSA and *word2vec* are applied in *DeleSmell*. LSA is used to calculate the similarity between methods or classes to evaluate the cohesiveness of the model. *Word2vec* selects the phrases that represent the semantics of code blocks through *Skip-gram* and maps them into vectors for model training.

4.3.2. Structural features extraction

Multiple metrics are selected to detect a kind of code smell since each metric has a different role and weight. We employ iPlasma [13] to extract structural metrics from the method or the class in a specific project. In addition to the metrics of a method, we also use the metrics of project, package, and class. We select 24 metrics for BC code smell and 21 metrics for BM code smell. The chosen metrics are presented in Table 4. These metrics (such as average methods weight (AMW), FANIN, FANOUT, etc.) are commonly employed in the field of code smell detection.

4.3.3. Features integration

The vector extracted from each item is taken as a sample of the dataset. The format of a vector is defined in Eq. (6).

$$< m_1, m_2, \ldots, m_n, s_1, s_2, s_3, ACSM > \quad (6)$$

where m_i represents the ith metrics, n represents the number of selected code metrics, s_j denotes the jth word vector extracted from the source code by *word2vec*, and *ACSM* represents the average conceptual similarity.

The BM dataset has more than 200,000 samples, while the BC dataset contains more than 22,000 samples. The final dataset contains approximately 38% positive samples.

4.4. Deep learning model

The model is critical to improving the performance of code smell detection. Inspired by Jiang's work [35], we propose a model with a CNN and GRU-attention in parallel and leverage an SVM as the classifier.

Fig. 5 presents the architectural overview of our model. The input is processed by a GRU-attention branch and CNN branch in parallel. The CNN branch consists of a feature extraction component followed by a classification component. The feature extraction component is composed of a set of hidden layers, including convolution, batch normalization, and dropout layers. The output of the last dropout layer is connected to the input of a densely connected network that consists of a stack of two dense layers. The reason why we employ a GRU is that it can save a considerable amount of time in the presence of numerous training samples. An attention mechanism is introduced in the GRU branch to learn the important features in the dataset while suppressing the interference of irrelevant information on the classification results. For an SVM, the kernel method can be utilized

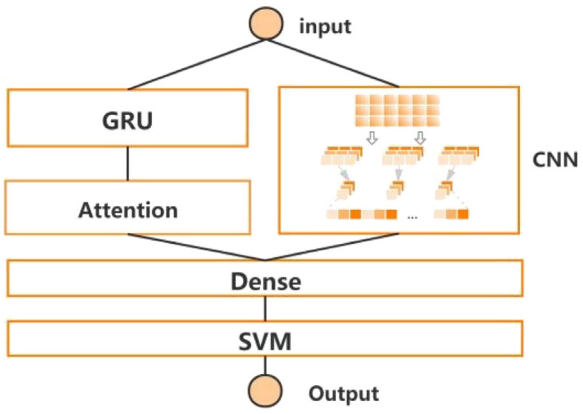


Fig. 5. Deep learning model of *DeleSmell*.

to map the nonlinear samples to high dimensional space and to identify the optimal hyperplane by maximizing the classification interval between two samples. The input of the last dense layer consists of the features concatenated by the GRU-attention branch and CNN branch and is connected to an SVM for the final classification. We tune the hyperparameters of the classifier by applying the *grid search* algorithm [36].

To avoid overfitting of the neural network, we employ certain optimization techniques. First, the function *ReLU* is selected as the activation function instead of the function *Sigmoid* in the convolution layers. Second, we penalize the loss function through L2 regularization at the dense layer. Generally, L2 regularization is better than L1 regularization if there is no particular concern for certain explicit features. Last, we use dropout layers with a ratio of 0.3. Additionally, we leverage *Adam* to optimize the defined model. The function *fit*() is used to train the model; its parameters include the data, labels, number of epochs, validation_data, and batch size. During the model training procedure, we set the epochs to 80 and the batch size to 16.

5. Evaluation

This section introduces the experimental setup and then presents the research questions and evaluation metrics. The experimental results are illustrated. For the reproducibility of the evaluation, all models and datasets are available at https://uzhangyang.github.io/research/delesmell.html.

5.1. Setup

All experiments are conducted on a workstation with a 2.6 GHz Intel Core i7 CPU and 16 GB main memory running 64-bit Windows 10. The Python version is 3.6, the TensorFlow version is 2.4.1 and the Keras version is 2.4.3.

Table 5 presents the configurations of all models.

5.2. Research questions

We evaluate the effectiveness of *DeleSmell* by answering the following research questions (RQ):

RQ1 Is *DeleSmell* more effective than ML-based approaches?

RQ2 Is our model more applicable than existing deep learning models for code smell detection?

RQ3 How effective is *DeleSmell* in detecting BC/BM code smell compared with previous works?

Table 4
Selected metrics.

BC					BM				
Size	Complexity	Cohesion	Coupling	Others	Size	Complexity	Coupling		Others
LOC	AMW	TCC	ATDF	NOPA	LOC	ALD	ATFD	FANIN	LAA
NAS	WMC	GREEDY	CC	DIT	NOEU	CYCLO	CC	FANOUT	CALIN
NOA	WOC		CM	BOVM		MAXNESTING	CCL	FANOUTCLASS	CEXT
NOAM			CBO	BUR		NOAV	CDISP	FDP	
NOM			DAC	CRIX		NOLV	CINT	NOP	
NDU			FANOUT				CM		
NOD			FDP						

Table 5
Configuration of models.

Model	Hyper-parameter	Value
SVM	degree	3
	tol	0.001
	cache_size	200
Random forest	n_estimators	100
Naive Bayes	alpha	1
Decision tree	max_features	3
Logistic	penalty	l2
	tol	0.0001
	max_iter	100
KNN	n_neighbors	3
BP	Number of units(Dense)	{64,128}
GRU	GRU units	{1,2}
ResNet	Filters in convolution layer	{64, 128, 256, 512}
	Kernel size in convolution layer	{3}
DeepFM	Dimensionality of embedding layer	{256}
	Number of units(Dense)	{32,64,128,256}
	LSTM units	{12}
CNN+LSTM	Filters in convolution layer	{32, 64, 128, 256, 512}
	Kernel size in convolution layer	{6, 10}
	LSTM units	{1,2}
	Number of units(Dense)	{128,256}
DeleSmell	Filters in convolution layer	{32, 64, 128, 256, 512}
	Kernel size in convolution layer	{6, 10}
	GRU units	{1,2}
	Number of units(Dense)	{128,256}

RQ4 How useful is our refactoring tool in improving the quality of the dataset?

RQ5 Is the performance of the model trained by a dataset with structural and semantic features better than that with barely structural or semantic features?

RQ6 How useful is LSA in improving the precision of code smell detection?

RQ7 How effective is *DeleSmell* in real-world projects? Is the generalization of *DeleSmell* good?

5.3. Evaluation metrics

We evaluate the effectiveness of *DeleSmell* by calculating the accuracy, precision, recall, and F-measure. These metrics can be computed by Eqs. (7)–(10).

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN} \tag{7}$$

$$\text{Precision} = \frac{TP}{TP + FP} \tag{8}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{9}$$

$$F - \text{measure} = 2 * \frac{\text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}} \tag{10}$$

where true positive (TP) indicates the number of positive samples correctly classified as positive, true negative (TN) indicates the number of negative samples correctly classified as negative, false-positive (FP) indicates the number of negative samples incorrectly classified as positive, and false negative (FN) indicates the number of positive samples incorrectly classified as negative.

5.4. Results

This section presents the experimental results.

5.4.1. Results for RQ1

To answer RQ1, we compare the performance of *DeleSmell* against that of six ML models: SVM [37], random forests [38], naive bayes [39], decision trees [40], logistic regression [41], and K-nearest neighbors (KNN) [42]. All ML models are available in *sklearn*. We conduct a controlled environment with 10-fold cross-validation.

Table 6 presents the accuracy, precision, recall, and F-measure of each ML model in BC/BM code smells detection. The results of *DeleSmell* have the best performance in all models. For BC code smell, the F-measure of *DeleSmell* is 3.33% (=98.05%−94.72%) higher than that of KNN. The F-measure presents better performance in both precision and recall. The situation is totally different when comparing the decision tree and *DeleSmell*. The recall and F-measure of *DeleSmell* are 97.13% and 98.05%, respectively,

Table 6
Results of RQ1.

Model	BC				BM			
	Accuracy	Precision	Recall	F-measure	Accuracy	Precision	Recall	F-measure
SVM	48.31%	51.01%	73.58%	59.07%	49.63%	65.77%	58.49%	62.19%
Random forest	88.19%	91.21%	84.52%	87.72%	72.48%	78.12%	62.45%	69.40%
Naive Bayes	85.27%	91.72%	77.57%	84.04%	61.79%	85.20%	78.53%	81.73%
Decision tree	92.26%	**99.38%**	85.05%	91.65%	68.60%	66.44%	75.16%	70.53%
Logistic	88.69%	89.36%	87.83%	88.59%	77.18%	81.38%	70.46%	75.53%
KNN	94.63%	93.18%	96.31%	94.72%	92.80%	88.10%	**98.96%**	93.22%
DeleSmell	**97.09%**	98.98%	**97.13%**	**98.05%**	**98.51%**	**99.01%**	98.62%	**98.81%**

which are 12.08% (=97.13%−85.05%) and 6.4% (=98.05%−91.65%) higher, respectively, than those of the decision tree. However, the precision of the decision tree is 0.4% (=99.38%−98.98%) higher than that of *DeleSmell*. The decision tree obtains a higher precision at the cost of a significant reduction in the recall. The SVM has the lowest precision, recall, and F-measure. These metrics of *DeleSmell* are 47.97% (=98.98%−51.01%), 23.55% (=97.13%−73.58%), and 38.98% (=98.05%−59.07%) higher than those of the SVM.

For the BM code smell, the precision of the random forest is 78.12%, while that of *DeleSmell* is 99.01%. The F-measure of random forest is 29.41% (=98.81%−69.4%) lower than that of *DeleSmell*. The accuracy of *DeleSmell* is 36.72% (=98.51%−61.79%) higher than that of naive bayes. In addition, *DeleSmell* outperforms the decision tree by improving F-measure from 70.53% to 98.81%, and recall from 75.16% to 98.62%. Compared with other machine learning models, the recall of KNN is 98.96%, which is 0.34%(=98.96%−98.62%) higher than that of *DeleSmell*. However, the precision of KNN is 10.91% (=99.01%−88.1%) lower than that of *DeleSmell*. KNN obtains a higher recall at the cost of a significant reduction in precision.

The experimental results show that *DeleSmell* significantly outperforms machine-learning-based approaches in detecting BC/BM code smells.

5.4.2. Results for RQ2

To answer RQ2, we compare the model of *DeleSmell* against existing deep learning models, including back propagation (BP) [43], GRU [44], ResNet [45], CNN [46], and DeepFM [47]. To make the comparison fair, all models are trained by the same dataset generated with the refactoring tool.

The evaluation results are tabulated in Table 7. For BC code smell, the precision, recall, and F-measure of *DeleSmell* are 15.46% (=98.98%−83.52%), 13.62% (=97.13%−83.51%), and 14.54% (=98.05%−83.51%) higher than those of BP. *DeleSmell* also outperforms the GRU by improving precision from 87.01% to 98.98%, recall from 81.50% to 97.13%, and F-measure from 84.21% to 98.05%. *DeleSmell* improves the F-measure by 5.29% (=98.05%−92.76%) compared with ResNet. Furthermore, the model improves recall by 2.59% (=97.13%−94.54%) and obtains 2.55% (=97.09%−94.54%) accuracy improvement compared with CNN+LSTM.

For BM code smell, *DeleSmell* dramatically improves precision by 18.5% (=99.01%−80.51%), recall by 18.5% (=98.62%−80.12%), and F-measure by 18.5% (=98.81%−80.31%) compared with BP. The precision, recall, and F-measure of the GRU are 85%, 84.52%, and 84.76%, which are 14.01% (=99.01%−85%), 14.1% (=98.62%−84.52%), and 14.05% (=98.81%−84.76%) lower, respectively, than those of *DeleSmell*. *DeleSmell* outperforms ResNet by improving the accuracy from 94.29% to 98.51%, recall from 94.29% to 98.62%, and F-measure from 94.29% to 98.81%.

We note that DeepFM has a high precision (97.38%) in BC code smell detection but at the cost of a significant reduction in recall (79.02%). After analyzing the confusion matrix, we discover that DeepFM needs to be improved in learning positive samples (smelly code) and is not suitable for code smell detection. Furthermore, we observe that the code smell detection results at different granularity levels perform differently in other deep learning models.

We conclude from these results that *DeleSmell* obtains high accuracy in detecting code smells at different granularity levels.

5.4.3. Results for RQ3

To answer RQ3, we compare *DeleSmell* against existing detection approaches for BC/BM code smells, such as MARS [11] and the CNN [9].

The evaluation results are presented in Table 8. For BC code smell, *DeleSmell* improves precision by 2.65% (=98.98%−96.33%), recall by 2.06% (=97.13%−95.07%), and F-measure by 2.35% (=98.05%−95.7%) compared with the CNN. The precision and F-measure of MARS are 2.78% (=98.98%−96.2%) and 2.21% (=98.05%−95.84%) lower than those of *DeleSmell*.

For BM code smell, the F-measure of CNN is 4.78% (=98.81%−94.03%) lower than that of *DeleSmell*. The recall and precision of *DeleSmell* are 5.51% (=98.62%−93.11%) and 4.03% (=99.01%−94.98%) higher, respectively, than those of the CNN. Compared with MARS, *DeleSmell* improves the F-measure by 2.98% (=98.81%−95.83%), recall by 2.8% (=98.62%−95.82%) and precision by 2.81% (=99.01%−96.2%). Furthermore, *DeleSmell* improves the accuracy by 4.41% (=98.51%−94.1%) and 2.32% (=98.51%−96.19%) compared with the CNN and MARS, respectively.

Compared with existing code smell detection approaches, *DeleSmell* achieves a remarkable improvement. We note that the performance of MARS on our dataset is slightly lower than its performance reported in [11]. A possible reason is that the number of features extracted by MARS is less than that extracted by DeleSmell. When MARS is trained by this dataset, the performance decreases, which motivates us to continue this work.

5.4.4. Results for RQ4

To answer RQ4, we generate two datasets to train the model. The first dataset is built by the iPlasma [13] and SMOTE algorithms without any interference from the refactoring tool, while the second dataset is constructed by the refactoring tool. LSA is leveraged in all datasets to analyze the semantic-related features.

The experimental results are summarized in Table 9. When detecting BC code smell, the performance of the model trained by the dataset with refactoring is significantly improved, with the exception of accuracy. Although the accuracy of the model trained by the dataset without refactoring is 1.42% (=98.51%−97.09%) higher than that of the model trained by the dataset with refactoring, the precision, recall, and F-measure are lower than those metrics obtained by the dataset with refactoring.

For BM code smell, the performance of the model trained by the dataset with refactoring significantly outperforms that trained by the original dataset. Specifically, the dataset with refactoring improves accuracy by 15.82% (=98.51%−82.69%) and precision by 4.27% (=99.01%−94.74%). The recall and F-measure of the model trained by the dataset with refactoring are 33.57% (=98.62%−65.05%) and 21.67% (=98.81%−77.14%) higher than those of the model trained by the dataset without refactoring.

Table 7
Results of RQ2.

Model	BC				BM			
	Accuracy	Precision	Recall	F-measure	Accuracy	Precision	Recall	F-measure
BP	83.55%	83.52%	83.51%	83.51%	86.34%	80.51%	80.12%	80.31%
GRU	86.09%	87.01%	81.50%	84.21%	83.66%	85.00%	84.52%	84.76%
ResNet	91.41%	95.94%	91.41%	92.76%	94.29%	94.59%	94.29%	94.29%
DeepFM	90.96%	97.38%	79.02%	87.25%	85.69%	82.53%	89.13%	85.70%
CNN+LSTM	94.54%	94.54%	94.54%	94.54%	94.54%	94.53%	95.03%	94.39%
DeleSmell	**97.09%**	**98.98%**	**97.13%**	**98.05%**	**98.51%**	**99.01%**	**98.62%**	**98.81%**

Table 8
Results of RQ3.

Approach	BC				BM			
	Accuracy	Precision	Recall	F-measure	Accuracy	Precision	Recall	F-measure
CNN [9]	95.52%	96.33%	95.07%	95.70%	94.10%	94.98%	93.11%	94.03%
MARS [11]	95.66%	96.20%	95.66%	95.84%	96.19%	96.20%	95.82%	95.83%
DeleSmell	**97.09%**	**98.98%**	**97.13%**	**98.05%**	**98.51%**	**99.01%**	**98.62%**	**98.81%**

Table 9
Results of RQ4.

	BC		BM	
	Without refactoring	With refactoring	Without refactoring	With refactoring
Accuracy	**98.51%**	97.09%	82.69%	**98.51%**
Precision	85.62%	**98.98%**	94.74%	**99.01%**
Recall	55.73%	**97.13%**	65.05%	**98.62%**
F-measure	67.51%	**98.05%**	77.14%	**98.81%**

With this evidence, we conclude that the technique of oversampling without validation probably leads to low recall. In addition, the technique of generating real positive samples with refactoring is useful for improving the quality of the dataset.

5.4.5. Results for RQ5

We compare the performance when different modalities are utilized for model training to answer RQ5. In this experiment, we built three types of datasets for BC and BM code smell: a dataset with only semantic features, a dataset with only structural features, and a dataset with both modalities of features. To make the comparison fair, all datasets are trained by the same model which has been described in Section 4.4.

Table 10 provides a summary of the experimental results. For BC code smell detection, the precision and F-measure of the dataset with only semantic features are decreased by 7.27% (=98.98%−91.71%) and 15.1% (=98.05%−82.95%), respectively, compared with the dataset with both modalities of features. Similarly, the corresponding values for the dataset with only structural features are 95.86% and 95.96%, which are 3.12%(=98.98%−95.86%) and 2.09% (=98.05%−95.96%) lower, respectively, than those of the dataset with both modalities of features.

For BM code smell, the F-measure of the dataset with only semantic features and structural features decreases by 18.34% (=98.81%−80.47%) and 1.78% (=98.81%−97.03%), respectively, compared with the dataset using both modalities of features. The dataset with only semantic features reduces precision and recall by 6.89% (=99.01%−92.12%) and 27.19% (=98.62%−71.43%), respectively, while the dataset with only structural features reduces precision and recall by 0.55% (=99.01%−98.46%) and 2.98% (=98.62%−95.64%), respectively.

With this evidence, we conclude that the model trained only by a single modality of features cannot achieve the required accuracy of code smell detection. The dataset with both modalities of features can significantly increase the performance of the model for code smell detection.

5.4.6. Results for RQ6

To evaluate the effectiveness of LSA for code smell detection, we compare the precision of one model trained by a dataset with LSA against that of another model trained by a dataset without LSA. In addition to the BC/BM code smell dataset, we build another two datasets for data class (DC) and shotgun surgery (SS) code smells. We note that SMOTE is leveraged to balance positive and negative samples in the DC/SS dataset. The intention of building the DC/SS code smell dataset is to evaluate the effectiveness of LSA without any interference from our refactoring tool. We also use these datasets to evaluate the effectiveness of *DeleSmell* in real-world projects (in RQ7).

The precision of the neural network model trained by four datasets is shown in Table 11. The dataset with LSA obtains higher precision than that without LSA. The precision is improved by 3.96% (=93.85%−89.89%) and 2.49% (=98.21%−95.72%) for BC and BM code smell detection, respectively. For DC/SS code smells, the precision of the dataset with LSA is 1.89% (=95.32%−93.43%) and 0.12% (=97.64%−97.52%) higher, respectively, than that without LSA. Note that the precision is improved for BC/BM code smells but only slightly improves for DC/SS code smells detection. A possible reason is that the ACSM calculated by LSA represents the cohesion of the source code, which is the main metric for BC/BM code smells. In contrast, cohesion is not an important metric for detecting DC/SS code smells. As a result, we believe that the ACSM calculated by LSA for semantic analysis can well represent the cohesion of the source code, help identify code smells, and is more suitable for BC/BM code smell detection.

5.4.7. Results for RQ7

To answer RQ7, we select 5 real-world projects to evaluate the performance of *DeleSmell* and to check whether the generalization of *DeleSmell* is good. We extract both structural features and semantic features from these projects and generate the dataset for prediction.

The experimental results are presented in Table 12. For BC code smell, *DeleSmell* achieves up to 99.65% accuracy in project *MiniTwit* [48]. The accuracy of other projects ranges from 96.00%

Table 10
Results of RQ5.

Features	BC				BM			
	Accuracy	Precision	Recall	F-measure	Accuracy	Precision	Recall	F-measure
With semantic features	91.61%	91.71%	75.12%	82.95%	92.37%	92.12%	71.43%	80.47%
With structural features	96.07%	95.86%	96.07%	95.96%	98.32%	98.46%	95.64%	97.03%
DeleSmell with both features	**97.09%**	**98.98%**	**97.13%**	**98.05%**	**98.51%**	**99.01%**	**98.62%**	**98.81%**

Table 11
Results of RQ6.

	BC		BM		DC		SS	
	without LSA	with LSA	without LSA	with LSA	without LSA	with LSA	without LSA	with LSA
Precision	89.89%	93.85%	95.72%	98.21%	93.43%	95.32%	97.52%	97.64%

Table 12
Results of RQ7.

Project	BC				BM			
	Accuracy	Precision	Recall	F-measure	Accuracy	Precision	Recall	F-measure
fop-core	96.34%	98.93%	96.35%	97.62%	98.68%	98.44%	98.68%	98.56%
JAdventure	97.06%	97.15%	97.01%	97.08%	97.78%	98.84%	97.89%	98.36%
MiniTwit	**99.65%**	**99.03%**	**99.15%**	**99.09%**	96.74%	96.73%	94.73%	95.72%
commons-lang	96.35%	98.72%	93.35%	95.96%	99.32%	**99.13%**	99.32%	99.22%
redomar	96.00%	95.62%	95.03%	95.32%	**99.50%**	99.01%	**99.50%**	**99.25%**
Average	97.08%	97.89%	96.18%	97.02%	98.40%	98.43%	98.02%	98.22%
Project	DC				SS			
	Accuracy	Precision	Recall	F-measure	Accuracy	Precision	Recall	F-measure
fop-core	91.73%	91.37%	92.16%	91.76%	94.33%	97.96%	94.33%	96.11%
JAdventure	92.50%	85.56%	92.50%	88.90%	96.37%	92.31%	92.31%	92.31%
MiniTwit	89.22%	91.13%	89.22%	90.16%	**98.00%**	95.50%	**98.00%**	96.73%
commons-lang	**97.25%**	**94.58%**	**97.25%**	**95.90%**	97.61%	99.52%	97.61%	**98.56%**
redomar	90.00%	81.00%	90.00%	85.26%	97.20%	**99.65%**	97.20%	98.41%
Average	92.14%	88.73%	92.23%	90.40%	96.70%	96.99%	95.89%	96.42%

to 97.06%. Project *redomar* [49] has the lowest accuracy of all projects, possibly because it has a small number of samples. For project *commons-lang* [50], the precision reaches 98.72%, but the recall is only 93.35%. For BM code smell, *DeleSmell* obtains up to 99.50% accuracy and recall. For projects *commons-lang* and *redomar*, the values of all metrics exceed 99%. However, when detecting *MiniTwit*, the recall is only 94.73%. The precision, recall, and F-measure of project *fop-core* [51] are 98.44%, 98.68%, and 98.56%, while those of project *JAdventure* are 98.84%, 97.89%, and 98.36%, respectively. For DC code smell, project *commons-lang* with the highest accuracy (97.25%) only obtains a 95.90% F-measure. For other projects, the recall and F-measure of project *fop-core* reach 92.16% and 91.76%, while those of project *MiniTwit* are only 89.22% and 90.16%. For SS code smell, the maximum accuracy and precision reach 98% and 99.65%, respectively. The precision of *fop-core* is 97.96%, but the recall is only 94.33%. The mean value of precision and F-measure for all projects achieve 96.99% and 96.42%, respectively.

DeleSmell achieves an average F-measure of 97.02% for BC code smell detection and an average F-measure of 98.22% for BM code smell detection. For DC code smell, the mean values of recall and F-measure are 3.95% (=96.18%−92.23%) and 6.62% (=97.02%−90.4%) lower, respectively, than those of BC code smell. The mean values of accuracy and recall for SS code smell are 1.7% (=98.4%−96.7%) and 2.13% (=98.02%−95.89%) lower, respectively, than those of BM code smell. It is probable that the datasets are constructed in a different way. The model trained by the dataset with our refactoring tool achieves a better performance in real-world project detection.

The experimental results show that *DeleSmell* is effective in real-world projects. When it is applied to other projects, the generalization of *DeleSmell* is good.

5.5. Threats to validity

This section outlines the threats to the validity of our results.

The first threat to the validity of our results is that this empirical study is focused on datasets created from source code written in Java and may not apply to other languages. Metrics and design styles vary by language, which may cause a lack of generalization of the results.

The second threat to the validity is that the ACSM calculated by LSA for semantic analysis represents the cohesion of the source code, which is an important metric for BC/BM code smell detection. This threat may have no significant impact on other code smells. However, our semantic analysis approach still has some implications for future research on code smell detection.

The third threat to the validity of our results is that we barely validate *DeleSmell* on four kinds of code smells and not on other code smells. The results may differ when applying our approach to other types of code smell detection. However, we perform many experiments by answering the research questions. It seems very likely that our conclusions are still useful and contribute to related research on code smell detection.

6. Conclusions

This paper presents a novel approach called *DeleSmell* for code smell detection. To increase the number of positive samples, a refactoring tool is developed as an *Eclipse* plugin to automatically generate items by static analysis for BC/BM detection instead of *SMOTE* algorithm. We extract both structural and semantic features from the items to generate the dataset via LSA and $word2vec$. To analyze the cohesion of the items, we perform LSA

on the code to calculate the ACSM. The training dataset is used as the input of a deep learning model which is composed of a CNN branch and a GRU-Attention branch with a final classification process performed by an SVM. To evaluate the effectiveness of *DeleSmell*, we conduct a bunch of experiments by answering seven research questions.

The experimental results show that *DeleSmell* improves the accuracy of BC/BM code smell detection by up to 4.41% compared with existing approaches, demonstrating the effectiveness of our approach.

Our future work involves applying this approach to more types of code smells and further identifying the severity of code smell to recommend a proper refactoring sequence for developers.

CRediT authorship contribution statement

Yang Zhang: Proposing the idea, Experimental analysis, Writing – review & editing. **Chuyan Ge:** Conducting experimentation, Writing – original draft. **Shuai Hong:** Developing a refactoring tool to help generate the dataset, Writing – review & editing. **Ruili Tian:** Writing – review & editing. **Chunhao Dong:** Building the deep learning model, Writing – review & editing. **Jingjing Liu:** Building the deep learning model, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The authors also gratefully acknowledge the insightful comments and suggestions of the reviewers, which have improved the presentation. This work is partially supported by National Nature Science Foundation of China under grant No. 61440012 and by Innovation Foundation of Hebei Province, China under grant No. CXZZSS2022081.

References

[1] F. Palomba, D. Tamburri, F. Arcelli Fontana, R. Oliveto, A. Zaidman, A. Serebrenik, Beyond technical aspects: How do community smells influence the intensity of code smells? IEEE Trans. Softw. Eng. 47 (1) (2021) 108–129.
[2] E.V. Emden, L. Moonen, Java quality assurance by detecting code smells, in: Working Conference on Reverse Engineering, 2002, pp. 97–106.
[3] N. Moha, Y.G. Gueheneuc, L. Duchien, A.F. Le Meur, DECOR: A method for the specification and detection of code and design smells, IEEE Trans. Softw. Eng. 36 (1) (2010) 20–36.
[4] W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, A. Ouni, A cooperative parallel search-based software engineering approach for code-smells detection, IEEE Trans. Softw. Eng. 40 (9) (2014) 841–861.
[5] T. Guggulothu, S.A. Moiz, Code smell detection using multi-label classification approach, Softw. Qual. J. 28 (9) (2020) 1063–1086.
[6] A. Kaur, S. Jain, S. Goel, A support vector machine based approach for code smell detection, in: 2017 International Conference on Machine Learning and Data Science, MLDS, 2017, pp. 9–14.
[7] F. Khomh, S. Vaucher, Y.G. Gueheneuc, H. Sahraoui, BDTEX: A GQM-based Bayesian approach for the detection of antipatterns, J. Syst. Softw. 84 (4) (2011) 559–572.
[8] T. Sharma, D. Spinellis, A survey on software smells, J. Syst. Softw. 138 (APR.) (2018) 158–173.
[9] A.K. Das, S. Yadav, S. Dhal, Detecting code smells using deep learning, in: TENCON 2019 - 2019 IEEE Region 10 Conference, TENCON, 2019, pp. 2081–2086.
[10] T. Sharma, V. Efstathiou, P. Louridas, D. Spinellis, Code smell detection by deep direct-learning and transfer-learning, J. Syst. Softw. 176 (4) (2021) 110936.
[11] Y. Zhang, C. Dong, MARS: Detecting brain class/method code smell based on metric–attention mechanism and residual network, 2021,
[12] N.V. Chawla, K.W. Bowyer, L.O. Hall, W.P. Kegelmeyer, SMOTE: Synthetic minority over-sampling technique, J. Artificial Intelligence Res. 16 (1) (2002) 321–357.
[13] C. Marinescu, R. Marinescu, P.F. Mihancea, D. Ratiu, R. Wettel, Iplasma: An integrated platform for quality assessment of object-oriented design, in: IEEE International Conference on Software Maintenance-Industrial and Tool Volume, 2005, pp. 77–80.
[14] M.A. Sai, D. Dong, Detection of large class based on latent semantic analysis, Comput. Sci. 44 (z6) (2017) 495–498.
[15] T. Mikolov, K. Chen, G. Corrado, J. Dean, Efficient estimation of word representations in vector space, Comput. Sci. (2013) 1301–3781, URL http://arxiv.org/abs/1301.3781.
[16] M. Shahidi, M. Ashtiani, M. Zakeri-Nasrabadi, An automated extract method refactoring approach to correct the long method code smell, vol. 187, 2022, 111221.
[17] F. Pecorelli, S. Lujan, V. Lenarduzzi, F. Palomba, A.D. Lucia, On the adequacy of static analysis warnings with respect to code smell prediction, Empir. Softw. Eng. 27 (3) (2022) 64.
[18] R. Yedida, T. Menzies, How to improve deep learning for software analytics (a case study with code smell detection), in: IEEE/ACM 19th International Conference on Mining Software Repositories,MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022, 2022, pp. 156–166.
[19] Z. Huang, Z. Shao, G. Fan, H. Yu, K. Yang, Z. Zhou, Hbsniff : A static analysis tool for java hibernate object-relational mapping code smell detection, Sci. Comput. Program. 217 (2022) 102778.
[20] B. Rga, A. Sks, A novel metric based detection of temporary field code smell and its empirical analysis - ScienceDirect, (ISSN: 1319-1578) 2021, URL https://www.sciencedirect.com/science/article/pii/S1319157821003050,
[21] M. Fokaefs, N. Tsantalis, E. Stroulia, A. Chatzigeorgiou, JDeodorant: identification and application of extract class refactorings, 2011, pp. 1037–1039.
[22] L. Amorim, E. Costa, N. Antunes, B. Fonseca, M. Ribeiro, Experience report: Evaluating the effectiveness of decision trees for detecting code smells, in: IEEE International Symposium on Software Reliability Engineering, 2015, pp. 261–269.
[23] F.A. Fontana, M. Zanoni, Code smell severity classification using machine learning techniques, Knowl.-Based Syst. 128 (C) (2017) 43–58.
[24] B. Aaa, A. Ha, Code smell detection using feature selection and stacking ensemble: An empirical investigation, Inf. Softw. Technol. 138 (2021) 106648.
[25] H. Liu, J. Jin, Z. Xu, Y. Bu, L. Zhang, Deep learning based code smell detection, IEEE Trans. Softw. Eng. 47 (9) (2021) 1811–1837.
[26] F. Pecorelli, D.D. Nucci, C.D. Roover, A.D. Lucia, On the role of data balancing for machine learning-based code smell detection, in: The 3rd ACM SIGSOFT International workshop, 2019, pp. 19–24.
[27] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, A.D. Lucia, The scent of a smell: An extensive comparison between textual and structural smells, IEEE Trans. Softw. Eng. 44 (10) (2018) 977–1000.
[28] M. Lanza, R. Marinescu, Object-Oriented Metrics in Practice - using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems, Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems, Object-Oriented Metrics in Practice, 2006.
[29] M. Fowler, Refactoring Improving the Design of Existing Code, Addison-Wesley, 1999, URL http://martinfowler.com/books/refactoring.html.
[30] JAdventure https://github.com/Progether/JAdventure.git.
[31] S.C. Deerwester, S.T. Dumais, T.K. Landauer, G.W. Furnas, R.A. Harshman, Indexing by latent semantic analysis, J. Am. Soc. Inf. Sci. 41 (6) (1990) 391–407.
[32] X. Guo, C. Shi, H. Jiang, Deep semantic-based feature envy identification, in: The 11th Asia-pacific symposium, 2019, pp. 19:1–19:6.
[33] H. Francescafallucc, F. Zanzotto, Inductive probabilistic taxonomy learning using singular value decomposition, Nat. Lang. Eng. 17 (PT.1) (2011) 71–94.
[34] Y. Goldberg, O. Levy, Word2vec explained: deriving Mikolov others,'s negative-sampling word-embedding method, 2014, CoRR abs/1402.3722 URL http://arxiv.org/abs/1402.3722.
[35] Y. Jiang, M. Jia, B. Zhang, L. Deng, Malicious domain name detection model based on CNN-gru-attention, 2021, pp. 1602–1607.
[36] J. Bergstra, Y. Bengio, Random search for hyper-parameter optimization, J. Mach. Learn. Res. 13 (1) (2012) 281–305, URL https://dl.acm.org/doi/10.5555/2503308.2188395.
[37] S. Amari, S. Wu, Improving support vector machine classifiers by modifying kernel functions, Neural Netw. 12 (6) (1999) 783–789.
[38] T. Luczak, B.G. Pittel, Componenets of random forests, Combin. Probab. Comput. 1 (1992) 35–52.
[39] I. Rish, An empirical study of the naive Bayes classifier, J. UCS 1 (2) (2001) 127.
[40] S.R. Safavian, D. Landgrebe, A survey of decision tree classifier methodology, IEEE Trans. Syst. Man Cybern. 21 (3) (1991) 660–674.

[41] C.M. Bishop, Pattern Recognition and Machine Learning (Information Science and Statistics), Pattern Recognition and Machine Learning (Information Science and Statistics), 2006, URL https://www.worldcat.org/oclc/71008143.
[42] Y. Liao, V.R. Vemuri, Use of K-nearest neighbor classifier for intrusion detection, Comput. Secur. 21 (5) (2002) 439–448.
[43] V. Bevilacqua, D. Daleno, L. Cariello, G. Mastronardi, Pseudo 2D hidden Markov models for face recognition using neural network coefficients, in: IEEE Workshop on Automatic Identification Advanced Technologies, 2007, pp. 107–111.
[44] K. Cho, B.V. Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, Y. Bengio, Learning phrase representations using RNN encoder-decoder for statistical machine translation, Comput. Sci. (2014) 1724–1734.
[45] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: 2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR, 2016, pp. 770–778.
[46] K. Fukushima, Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position, Biol. Cybernet. 36 (4) (1980) 193–202.
[47] H. Guo, R. Tang, Y. Ye, Z. Li, X. He, in: C. Sierra (Ed.), DeepFM: A Factorization-Machine based Neural Network for CTR Prediction, ijcai.org, 2017, pp. 1725–1731.
[48] MiniTwit https://github.com/eh3rrera/minitwit.
[49] redomar https://github.com/redomar/backend-redomar.
[50] commons-lang https://github.com/apache/commons-lang.
[51] fop-core https://github.com/Coder-VenkateshS/FOP-JAVA-Core-JAVA.