Fusion of deep convolutional and LSTM recurrent neural networks for automated detection of code smells

Anh Ho
anh.h190037@sis.hust.edu.vn
Hanoi University of Science and Technology
Hanoi, Vietnam

Anh M. T. Bui*
anhbtm@soict.hust.edu.vn
Hanoi University of Science and Technology
Hanoi, Vietnam

Phuong T. Nguyen
phuong.nguyen@univaq.it
Università degli studi dell'Aquila
67100 L'Aquila, Italy

Amleto Di Salle
amleto.disalle@unier.it
Università Europea di Roma
00163 Roma, Italy

Abstract

Code smells is the term used to signal certain patterns or structures in software code that may contain a potential design or architecture problem, leading to maintainability or other software quality issues. Detecting code smells early in the software development process helps prevent these problems and improve the overall software quality. Existing research concentrates on the process of collecting and handling dataset, then exploring the potential of utilizing deep learning models to detect smells, while ignoring extensive feature engineering. Though these approaches obtained promising results, the following issues need to be tackled: *(i)* extracting both structural and semantic features from the software units; *(ii)* mitigating the effects of imbalanced data distribution on the performance.

In this paper, we propose DeepSmells as a novel approach to code smells detection. To learn the complex hierarchical representations of the code fragment, we apply a deep convolutional neural network (CNN). Then, in order to improve the quality of the context encoding and preserve semantic information, long short-term memory networks (LSTM) is placed immediately after the CNN. The final classification is conducted by deep neural networks with weighted loss function to reduce the impact of skewed data distribution. We performed an empirical study using the existing code smell benchmark datasets to assess the performance of our proposed approach, and compare it with state-of-the-art baselines. The results demonstrate the effectiveness of our proposed method for all kinds of code smells with outperformed evaluation metrics in terms of F1 score and MCC.

ACM Reference Format:
Anh Ho, Anh M. T. Bui, Phuong T. Nguyen, and Amleto Di Salle. 2023. Fusion of deep convolutional and LSTM recurrent neural networks for automated detection of code smells. In *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering (EASE '23), June 14–16, 2023, Oulu, Finland.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3593434.3593476

*Corresponding author

1 INTRODUCTION

Fowler [8] initially coined the concept of *code smell* to indicate quality issues in code that could be further refactored to improve the maintainability of software systems. The presence of smells in code mainly comes from shortcut implementations, temporary workarounds [3], or sub-optimal design solutions to speed up the software development in the short term (a.k.a technical debts) [27]. Prior research has shown that while code smells do not impact the software system's functionality, they may limit its reusability and extensibility [26].

The impact of code smell would be reduced by applying a refactoring process [4], which seeks to reconstruct the internal software structure without altering its external behavior [8]. The refactoring process consists of three steps: *(i)* identifying code smells; *(ii)* applying adequate refactoring operations to correct them; and *(iii)* assuring the preservation of the system's functionalities [21]. It is, therefore, essential to identify code smells to increase the productivity of the software development process. In fact, early identification of code smell is crucial to lowering the cost of the refactoring process. Undoubtedly, automated code smell detection is desirable because manual methods are cumbersome and time-consuming.

A large body of work has been dedicated to detecting smells in source code. Traditional approaches rely on handcraft software metrics and employ heuristic evaluations to classify code snippets as *smelly* and *non-smelly* [15]. On the one hand, it is challenging to construct the optimal heuristic evaluation manually. On the other hand, using software metrics may lead to a potential classification bias. Indeed, code fragments with different behaviors can use the same metrics but provide a totally different code smell. ML approaches have been proposed for code smell detection to overcome these problems [14, 16]. In particular, in the last years, numerous studies applied deep learning techniques to detect code smells, including artificial neural networks (ANNs) [10], convolutional neural networks (CNNs) [6, 18, 29], recurrent neural networks (RNNs) [24]. Empirical results have demonstrated a higher accuracy rate comparing deep learning-based code smell detectors to classical machine learning approaches. However, there is still space for improvement. Indeed, most studies have considered source code as text and applied natural language processing methods to text mining. It is unarguable that source code differs from natural language due to the syntax difference between the two languages. Treating code as text and ignoring the semantics of underlying structures (e.g., nesting control) in source code may lead to the risk of not preserving the correct meaning. We believe that capturing the semantic structures of the source code plays an important role in detecting code smells.

This paper proposes DeepSmells, a novel approach to code smell detection built on top of cutting-edge deep learning techniques. First, to learn the complex hierarchical representations of code fragments, we apply a deep convolutional neural network (CNN). Unlike other state-of-the-art studies that employ word embedding techniques to represent source code identifiers, our work encodes the whole source code by tokenization indexing before sending it to the 1D-CNN model to learn the structured patterns. Then, to improve the context encoding quality and preserve semantic information, long short-term memory networks (LSTMs) are placed immediately after the CNN. Deep neural networks conduct the final classification with a weighted loss function to reduce the impact of skewed data distribution. We detect four types of code smell, including *Complex Method, Complex Conditional, Feature Envy*, and *Multifaceted Abstraction* using the datasets curated by Sharma et al. [24].

The main contributions of our work are briefly summarized as follows:

- A code smell detector, namely DeepSmells, built on top of cutting-edge deep learning techniques to deal with complex hierarchical representations of code fragments.
- An empirical evaluation using real-world dataset, and comparison with state-of-the-art baselines.
- The tool developed together with the metadata process in this paper has been published to facilitate future research.[*]

The paper is structured as follows. Section 2 reviews related studies. Section 3 presents the proposed approach built on top of deep neural networks. Section 4 elaborates on the datasets, settings, and metrics used to perform an empirical evaluation. Section 5 reports and analyzes the experimental results. Finally, Section 6 concludes the paper.

2 RELATED WORK

Many studies have been published in the literature on detecting code smells. Sharma et al. [26] proposed five categories of smell detection methods. Since our work focuses on code smell detection using machine learning techniques, we divided the related work into two subsections, i.e., *ML-based code detection* and *code smell detection through other techniques.*

2.1 ML-based smell detection

These approaches use *machine learning* techniques, including Bayesian [16], Support Vector Machine [14]. Although existing ML-based methods achieve encouraging prediction performance, they suffer from insufficient feature extraction, which requires additional research [26]. Therefore, deep learning techniques have been exploited in recent years to detect code smells. For example, Hadj-Kacem et al. [10] combined two learning techniques, i.e., deep auto-encoder and a neural network with one hidden layer, to detect four code smells. The approach has been evaluated using four datasets containing 74 open-source projects. Results show a high accuracy with more than 96% for precisions, recalls, and F1 scores concerning four code smells. Das et al. [6] used a convolutional neural network (CNN) to detect two code smells (Brain Class and Brain Method) using thirty open-source Java projects. Results show good accuracy for both Brain Class, and Brain Method smells. Liu et al. [18] employed a text embedding technique (i.e., word2vec) to represent code fragments before sending them to a CNN model for classification. The CNN model was utilized to extract essential features and reduce the dimensionality to boost the performance of the classification task. Zhang et al. [29] also adopted CNN techniques combined with the Support Vector Machine (SVM) algorithm to build a code smell detection. Sharma et al. [24] compared different deep learning techniques, i.e., CNN and RNN, along with autoencoder models, to detect four code smells (viz. complex method, complex conditional, feature envy, and multifaceted abstraction). Moreover, the authors analyzed whether deep learning models obtained from training in a programming language, i.e., C#, can be transferred to another language, i.e., Java. Results show that the model performance, i.e., RNN or CNN, depends on which code smell has to be detected. Concerning the transfer-learning, results demonstrate that it is feasible.

[*]https://github.com/EASE2023-DeepSmells/DeepSmells

2.2 Other approaches

The most common methods are *metric-based smell detection* which aim to identify a code smell through a formula combining a set of metrics. The formula applies filters and uses thresholds for the related metrics [19]. The metrics values are calculated using a source code model such as an Abstract Syntax Tree (AST) extracted from the source code. Marinescu [20] defined formulas for ten code smell using different metrics in evolving object-oriented systems. For example, the number of public attributes (NOPA) and the number of accessor methods (NOAM) metrics are combined to detect the data class smell, i.e., poor encapsulated classes. Macia et al. [2] identified smell for aspect-oriented programming based on metrics. The authors defined formulas for fifteen well-known and new code smells combining eight metrics and also defined the related thresholds. Moreover, the authors evaluated the smell detection strategies using three applications and seventeen releases. Vidal et al. [28] proposed a semi-automated approach to prioritize code smells. Moreover, the authors developed a tool named SpIRIT to identify ten code smells using well-known detection strategies where a rule defines the smell through a combination of metrics and related thresholds [17]. Many approaches have been applied to Java code smell, but a few investigated dynamic languages such as JavaScript and Python. For example, Fard and Mesbah [7] proposed a metric-based tool to detect thirteen JavaScript code smell using static and dynamic source code analysis. Chen et al. [5] proposed another exciting study to discover code smells in Python. The authors defined ten code smells and indicated a metric-based method with three filtering strategies to establish metric thresholds.

Rules/heuristic-based smell detection methods specify rules or heuristics and leverage to source code model and optionally metrics for detecting code and principally design smells [22, 27] if rules/heuristics are fulfilled. Sharma et al. [25] implemented Designite[†] to detect architectural, design, and implementation smells. It detects smell for Java and C# souce code and calculates metrics at different granularities, i.e., method-level, class-level, project-level, and solution-level. Moha et al. [22] proposed DECOR and DETEX to extract rule cards from natural text describing a smell using a defined domain-specific language. An algorithm is then generated from the rules to detect code and design smell. DETEX detects four design smells together with fifteen related code smells, and it has been validated using 11 open-source projects. Both metrics-based and rule-based/heuristic methods are simple to implement. However, the main challenge of these techniques concerns the definition of metrics with thresholds, and the results vary significantly among them.

History-based smell detection approaches exploit source code evolution to detect code smells through a detection model created using the evolved code [23]. Palomba et al. [23] proposed an approach named Historical Information for Smell deTection (HIST) to detect five code smells using evolution code information coming from version control systems. First, HIST extracts fine-grained changes from source code evolution and then uses different heuristics techniques depending on the code smell to detect. The approach has been evaluated to verify its accuracy in terms of precision and recall using 20 open-source projects concerning a manually-produced oracle. Results show that HIST's precision ranges between 72 and 86 percent, and the recall is between 58 and 100 percent. Fu and Shen [9] proposed another approach based on history changes of source code to detect three code smells, i.e., duplicated code, shotgun surgery, and divergent change. The approach extracts association rules from the change history and then uses heuristic algorithms to detect the three code smells.

Since only a small number of smells are connected to evolutionary changes, history-based techniques have a restricted range of applications. Because of this, history-based approaches cannot identify a source code item (such as a function or class) that has not necessarily changed over time to exhibit a smell.

[†]https://www.designite-tools.com/

Optimization-based smell detection approaches detect code smells using optimization algorithms based on calculated metrics. The algorithms can also use predefined examples to detect code smells. Kessentini et al. [15] presented an approach that runs in a parallel cooperative manner with different evolutionary algorithms to detect eight code smells. In particular, the approach uses two evolutionary algorithms executed in parallel to generate the best detection rules and detectors. Then a developer can exploit these rules and detectors to identify code smells on an implemented system.

3 PROPOSED SOLUTION

This section provides the details of our proposed approach for code smell detection. Before going into the details of the approach, we explain the idea behind it as follows. A big challenge of code smell detection is that different smells may be related to the same symptoms. For example, the violation of the single responsibility principle is the symptom of both *Multifaceted Abstraction* and *Divergent Change* [1]. As a result, a multi-label classification may not be sensitive for code smell detection. Similar to previous state-of-the-art studies, we also concentrate on single smell detection, which means that our proposed model works separately on different smell datasets for binary classification (i.e., *smelly* or *non-smelly*). Otherwise, the reasons for choosing four code smells (i.e., Complex Method, Complex Conditional, Feature Envy, and Multifaceted Abstraction) come from the intuitions of our approach. First, Complex Method and Complex Conditional smells are related to the implementation complexity. The ability to capture statement structures through the token-embedding of source code makes it sensitive to both the Complex Method smell, where nested conditional statements are frequently present and the Complex Conditional smell, where conditional expressions are lengthy or complicated. The CNN model is applied to learn features from various representations of code. Second, the two remaining smells, Feature Envy and Multifaceted Abstraction, are categorized as *design* smells, which tend to concern multiple methods or classes. In this context, an LSTM model that links sequential representations to learn features seems to work well. We also aim to assess different configurations of LSTM to investigate their ability to distinguish these smells.

Figure 1 depicts the overall architecture of DeepSmells. We divide the proposed code smell detector into two components: *(i)* the first component combines CNN and LSTM to extract essential features from the input source code; *(ii)* the second one, which is based on a fully connected network, runs as a binary classifier to map feature vectors generated from the first component to *smelly* or *non-smelly* labels. We implemented our proposed approach using the Pytorch framework.[‡] The code smell detection process consists of four phases as follows.

During the pre-processing phase, we perform various steps on each dataset including: *(i)* embedding source code by indexing code tokens; *(ii)* computing statistical information about the samples' length, and removing samples with a length exceeding one standard deviation from the mean; *(iii)* padding samples with the zero value to extend to the longest array input.

Subsequently, in the model-building phase, the pre-processed data is passed through convolution blocks to extract unique features from the source code automatically. Typically, many convolution blocks are employed to progressively learn more complex and abstract features from the input data, including high-level characteristics such as component relationships within the source code. However, when deeper networks can start converging, a degradation problem has been exposed: with the network depth increasing, accuracy gets saturated (which might be unsurprising) and then degrades rapidly [11]. In this case, two convolution blocks are utilized, which are configured as given below.

- The first convolution block consists of a 1D-CNN (`torch.nn.Conv1d`) with 16 filters with a kernel size k which specifies the length of 1D convolution window. We experiment different values of k (i.e., $k = 3, 4, 5, 6, 7$) to evaluate the effectiveness of the convolution layer. This layer employs a `ReLU` function, and the remaining parameters are kept as defaults. Following by a 1D-Batch Normalization (`torch.nn.BatchNorm1d`) to accelerate the network training and to reduce internal covariate shift [13]. The obtained feature map is then passed to a MaxPooling layer (`torch.nn-.MaxPool1d`) by a factor of size 3 to reduce the spatial dimension.
- The second convolution block is similar to the first one, excepting that we use 32 filters.

Once the output from the convolutional neural networks has been obtained, it is fed into the LSTM network to preserve the meaning and context of the data. The `torch.nn.LSTM` function is used to configure the LSTM network, with the input size of each LSTM unit being based on the initial size of the initial embedding source code vector. The number of LSTM units in the network is also set to 32 to match the 32 filters in the second convolution block. In addition, we implement BiLSTM to capture long-term dependencies in sequential data that span both forward and backward directions. BiLSTM consists of two separate layers of LSTM units, one for processing the input sequence in the forward direction and the other layer for the backward direction. The hidden state at a time step of the network is the concatenation of both the forward and backward hidden states, allowing the hidden states to capture future information. We use the same configuration as before but set the hyperparameter `bidirectional` to `True`.

Finally, the smell classification is performed by a deep neural network with one hidden layer using the activation function as `ReLU`. The output layer of one node uses the `Sigmoid` as the activation function. The number of hidden nodes is empirically turned to find the appropriate value for each dataset. However, when dealing with highly imbalanced data, the model's classification becomes biased toward the majority class, while the minority class is usually of greater importance. During neural network training, the cost function is the key to adjusting a neural network's weights to create a better-fitting machine learning model [12]. Thus, to solve the class imbalance problem in the dataset, we add sensitive weight into binary cross-entropy to adjust the importance of the minor class.

$$cost_i = \beta \hat{y}_i log(y_i) + (1 - \hat{y}_i) log(1 - y_i) \tag{1}$$

where β is the sensitive weight, y_i is the actual label of input x_i and y_i is the model's prediction for input x_i. To determine the optimal weight for our approach, we employ binary cross-entropy with β set to 1, as well as weighted binary cross-entropy with different weights (β = 2, 4, 8, 12, 32, and 84). By comparing the results, we choose the best hyperparameter for our proposed method.

In the training and testing phase, we trained our model using a mini-batch size of 128. The learning rate is set to 0.03. The training is done within 50 epochs using the `SGD` to optimize the weighted loss function.

4 EMPIRICAL SETTINGS

This section presents the evaluation conducted to study the performance of DeepSmells. Section 4.1 introduces the datasets used in the evaluation. Afterwards, the metrics are described in Section 4.2. Finally, we elaborate on the evaluation plan in Section 4.3.

4.1 Benchmark Datasets

We conduct experiments on the datasets curated by Sharma et al. [24]. Table 1 presents four code smell datasets including: *Complex Method, Complex Conditional, Feature Envy* and *Multifaceted Abstraction*. The total number of samples for all datasets is 416, 445 in which the total numbers of *smelly* (positive) and *non-smelly* (negative) instances are 20, 753 and 395, 692, respectively. The average imbalanced rate of all datasets is 5.24%.

The source code is examined at method-level for *Complex Method* and *Complex Conditional* datasets and at class-level for *Feature Envy* and *Multifaceted Abstraction* datasets. It can be seen that the final dataset, *Multifaceted*

[‡] https://pytorch.org

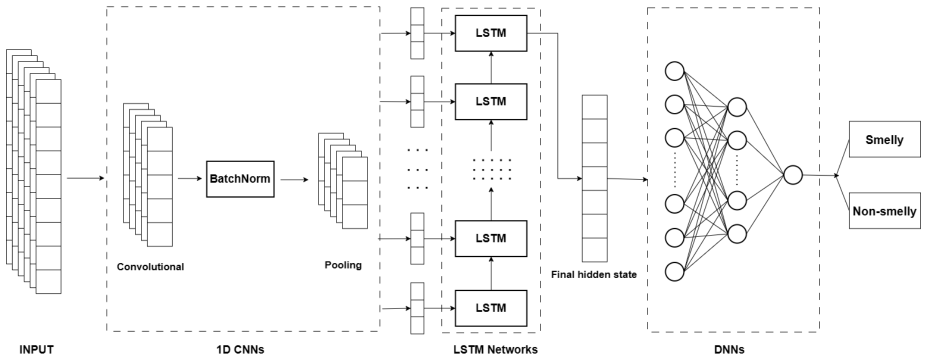

Figure 1: The overall DeepSmells architecture.

Table 1: Statistics of the datasets.

Smell	Smell Alias	# Positive	# Negative
Complex Method	CM	12,489	144,460
Complex Conditional	CC	6,186	149,767
Feature Envy	FE	1,788	51,260
Multifaceted Abstraction	MA	290	50,205

Abstraction has the lowest number of instances, which might rise challenges for deep learning models. We split each dataset into training and testing parts following a ratio of 70-30 (i.e., 70% for training and 30% for testing).

4.2 Evaluation Metrics

In order to evaluate the performance of our model, we employ widely-used evaluation metrics including *Precision (P), Recall (R), F1-Score (F1)* and *Matthews Correlation Coefficient (MCC)* [24, 26, 29].

4.2.1 Precision, Recall and F1-Score. The confusion matrix is the starting point to evaluate any classification models with four possible outcomes including *true positive* (TP), *true negative* (TN), *false positive* (FP) and *false negative* (FN).

Precision (P) which measures how many of the positive predictions made by the model are actually correct, is defined as:

$$P = \frac{TP}{TP + FP} \tag{2}$$

Recall counts the number of positive cases in the dataset that model can identify, and it is defined as:

$$R = \frac{TP}{TP + FN} \tag{3}$$

Finally, *F1-Score* represents the balance between *precision* and *recall* of the prediction model and is calculated as follows.

$$F1 = \frac{2 * P * R}{P + R} \tag{4}$$

4.2.2 MCC. This metric is useful when dealing with imbalanced classification. It measures the correlation between the predicted class and actual class which is scaled in the range $[-1, 1]$, where 1 represents a perfect prediction, and -1 shows a perfect negative correlation, i.e., the worst case. When MCC is equal to 0, the model shows a random prediction.

$$MCC = \frac{TP * TN - FP * FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{5}$$

4.3 Evaluation Plan

We compare the effectiveness of DeepSmells with three baseline models which have been introduced in a recent work of Sharma et al. [24]. They have investigated the efficiency of different variants of an auto-encoder model to detect code smells. The objective of an auto-encoder is to compress the source code and to learn salient information which is reflected in the reconstructed output [24]. Three architectures of auto-encoder have been studied and showed promising results, as follows.

- AE-Dense: The auto-encoder model employed denses for encoder and decoder layers
- AE-CNN: The auto-encoder model employed two CNN networks, the first one for encoder and the other for decoder
- AE-LSTM: Similar to AE-CNN but the CNNs are replaced by LSTM networks

The encoder and decoder layers are followed by a fully connected dense layer for classification. Different from their work, we aim to learn various code representations to understand patterns of both smelly and non-smelly samples through different convolutional filters and try to capture the semantic context through LSTM networks. We re-run all of these models on the same benchmark datasets (see Section 4.1) to compare to our model. In this paper, we aim to answer to the following research questions:

- **RQ$_1$**: *How do different DeepSmells configurations affect the prediction performance?* Among the considered network configurations, we study to find the one that brings the best prediction performance for detecting code smells.
- **RQ$_2$**: *How does DeepSmells perform compared to the baseline models?* We evaluate how well our proposed approach perform in comparison to three aforementioned baseline models.

5 RESULTS AND DISCUSSION

We present and analyze the experimental findings in order to respond to two research questions introduced in the previous section.

5.1 RQ$_1$: *How do different DeepSmells configurations affect the prediction performance?*

Our focus is to evaluate the efficiency of different network configurations with respect to the code smell prediction performance.

5.1.1 The effect of the convolution layers. The objective of the convolution layers is to learn the hierarchical representation within the source code. We then first investigate how the kernel size affects the capability to learn the local patterns of source code. For experimentation, we have considered various values for the kernel size of both convolution layers. In particular, we consider 5 scenarios in which the kernel size of each filter are adjusted to 3, 4, 5, 6 and 7, respectively. As can be seen in Figure 2, the proposed model attains the highest F1-Score and MCC measures with regard to the

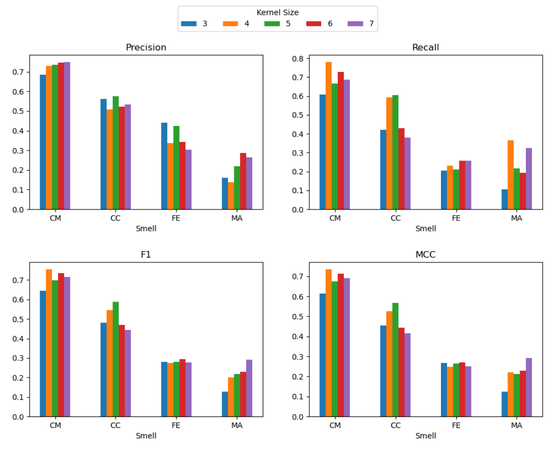

Figure 2: The impact of kernel size on the performance of DeepSmells.

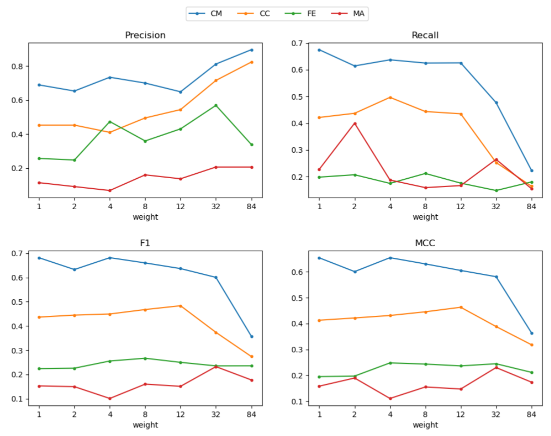

Figure 3: The impact of imbalanced weight on the performance of DeepSmells.

kernel size as 4 for the CM smell and 5, 6, 7 for the CC, FE and MA smells, respectively.

5.1.2 The effect of variant LSTM networks. Next, we evaluate the impact of two variants of LSTM model, uni-directional LSTM (that we call LSTM) and bi-directional LSTM (i.e., Bi-LSTM), on the performance of our proposed approach. Different to uni-directional LSTM architecture which allows to maintain long-range connections along a sequence of source code fragments, the bi-directional architecture takes into account the flow of inputs in both directions, leading to the capability to utilize information from both sides. Table 2 shows the performance of DeepSmells considering two architectures of LSTM. We run the experiments with the same configuration of CNN and DNN blocks (as described in Section 4).

Table 2: Comparison between the two LSTM architectures.

Smell	Model	Metric			
		P	R	F1	MCC
CM	DeepSmells	**0.7313**	**0.7786**	**0.7542**	**0.7341**
	DeepSmells-BiLSTM	0.7062	0.7599	0.7321	0.7104
CC	DeepSmells	0.5749	**0.6042**	0.5892	0.5684
	DeepSmells-BiLSTM	**0.5927**	0.5959	**0.5943**	**0.5731**
FE	DeepSmells	**0.3414**	**0.2581**	**0.2940**	**0.2686**
	DeepSmells-BiLSTM	**0.3414**	0.2334	0.2773	0.2520
MA	DeepSmells	0.2874	**0.2717**	**0.2793**	**0.2752**
	DeepSmells-BiLSTM	**0.3097**	0.2345	0.2669	0.2402

It is evident that regarding the three metrics including Recall, F1-Score and MCC, DeepSmells with LSTM performed slightly better than BiLSTM with respect to two smells, *Complex Method* and *Feature Envy*. In particular, for the CM code smell, DeepSmells with LSTM achieved a significant improvement of 8.68% in terms of recall, comparing to BiLSTM. The situation is totally different when dealing with two smells, *Complex Conditional* (CC) and *Multifaceted Abstraction* (MA). DeepSmells with BiLSTM outperforms LSTM in terms of recall, F1-Score and MCC. However, the LSTM model offers a better precision for two smells, CC and MA, but lower precision for two others, CM and FE, in comparison to BiLSTM.

5.1.3 The effect of the imbalanced weight. We finally investigate the impact of the imbalanced weight β (see Equation 1) on the prediction performance of DeepSmells. Figure 3 shows the performance of DeepSmells for seven values of β, i.e., $\beta = \{1, 2, 4, 8, 12, 32, 84\}$, with respect to four measures and four code smells. Considering the CM smell, given $\beta = 4$, DeepSmells achieves the best performance on recall, F1-Score and MCC. Regarding the CC smell, $\beta = 12$ gives the highest values for F1-Score and MCC. For the two remaining smells, FE and MA, the proposed model obtains the best performance in terms of F1-Score and MCC when $\beta = 32$. This may be due to their higher degree of imbalance comparing to the two first smells. Indeed, the proportion between positive and negative samples of CM, CC, FE and MA is 8.65%, 4.13%, 3.49%, 0.58%, respectively. In a nutshell, the empirical findings show that the more skewed the distribution is, the higher is the imbalance weight.

5.2 RQ$_2$: *How does DeepSmells perform compared to the baseline models?*

We conducted a performance comparison between our proposed approach and three baselines models presented in Section 4.3, including AE-Dense, AE-CNN and AE-LSTM. Table 3 shows the experimental results for each kind of smell. The table demonstrates that our proposed model outperforms the other models by all evaluation metrics. In particular, regarding the CM smell, DeepSmells yields an improvement of 13.50% and 14.69% with regards to F1-Score and MCC, respectively, in comparison with AE-Dense that is the best baseline model for this type of smell [24]. Furthermore, for the CC smell, our proposed model significantly outperformed AE-Dense by increasing accordingly about 37.19%, 0.67%, 13.50% and 14.69% on the value of precision, recall, F1-Score and MCC. However, considering the FE and MA smells, the situation is slightly different, i.e., the comparison between AE-CNN and DeepSmells shows diverse results. While DeepSmells achieves higher F1-Score and MCC values by 3.15% and 0.83%, respectively, the recall of AE-CNN is 28.04% better. Similarly, for the MA smell, the precision value is superior, but the recall value is much lower due to the high imbalance of this dataset, leading to a lower overall performance in terms of MCC, approximately 9.5%, compared to DeepSmells. Overall, *we can see that DeepSmells gains a superior prediction performance compared to the baselines by all the four types of code smells.*

5.3 Threats to validity

- **Internal validity.** This is related to the degree to which our evaluation resembles real-world scenarios. In the evaluation, we adopted existing datasets [24] manually curated and classified by humans. The quality of the curated data depends very much on the evaluators' expertise.

Table 3: Comparison with state-of-the-art baselines.

Smell	Model	Metric			
		P	R	F1	MCC
CM	AE-Dense	0.4834	0.6304	0.5472	0.5077
	AE-CNN	0.4721	0.5815	0.5211	0.4781
	AE-LSTM	0.4683	0.6146	0.5316	0.4905
	DeepSmells	**0.7313**	**0.7786**	**0.7542**	**0.7341**
CC	AE-Dense	0.1703	0.3874	0.2366	0.2108
	AE-CNN	0.1940	0.2759	0.2278	0.1933
	AE-LSTM	0.1797	0.3287	0.2324	0.2007
	DeepSmells	**0.5749**	**0.6042**	**0.5892**	**0.5684**
FE	AE-Dense	0.1703	0.3874	0.2366	0.2108
	AE-CNN	0.1572	**0.4925**	0.2384	0.2352
	AE-LSTM	0.1968	0.2537	0.2217	0.1968
	DeepSmells	**0.3414**	0.2581	**0.2940**	**0.2686**
MA	AE-Dense	0.0314	**0.7471**	0.0603	0.1351
	AE-CNN	0.0312	0.6782	0.0596	0.1272
	AE-LSTM	0.0328	0.4023	0.0606	0.0985
	DeepSmells	**0.2874**	0.2717	**0.2793**	**0.2752**

We anticipate that comments might be miss-classified, thus negatively impacting the overall prediction performance.

- **External validity.** This concerns the generalizability of the findings beyond the scope of this study. We attempted to mitigate the threats by evaluating with different experimental configurations to simulate real-world scenarios. The findings of our work might apply only to the considered datasets. For other datasets, we need additional empirical evidence before reaching a final conclusion.
- **Construct validity.** This is related to the experimental settings to compare DeepSmells with the baselines. Attempting to mitigate the threats, we simulated a real-world scenario where the systems are about to provide predictions based on the available labeled code smells. To guarantee a reliable comparison with the baselines, we used the original implementations made by their authors, leaving the internal design intact.

6 CONCLUSION AND FUTURE WORK

We proposed DeepSmells as a practical solution to code smells detection exploiting different deep learning techniques. An empirical evaluation on real-world datasets demonstrated the effectiveness of our proposed approach. We also proved that DeepSmells is more effective in classifying code smells compared to well-established baselines. For future work, we plan to improve the prediction engine by incorporating different deep learning techniques. We anticipate that the application of CodeBERT–a pre-trained model on sourcecode–may help boost up the overall effectiveness.

References

[1] Francesca Arcelli Fontana, Mika V Mäntylä, Marco Zanoni, and Alessandro Marino. 2016. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering* 21 (2016), 1143–1191.

[2] Isela Macia Bertran, Alessandro Garcia, and Arndt von Staa. 2010. Defining and Applying Detection Strategies for Aspect-Oriented Code Smells. In *24th SBES 2010, Salvador, Bahia, Brazil, September 27 - October 1, 2010*. IEEE Computer Society, 60–69. https://doi.org/10.1109/SBES.2010.14

[3] Gemma Catolino, Fabio Palomba, Francesca Arcelli Fontana, Andrea De Lucia, Andy Zaidman, and Filomena Ferrucci. 2020. Improving change prediction models with code smell-related information. *Empir. Softw. Eng.* 25, 1 (2020), 49–95. https://doi.org/10.1007/s10664-019-09739-0

[4] Alexander Chatzigeorgiou and Anastasios Manakos. 2010. Investigating the Evolution of Bad Smells in Object-Oriented Code. In *2010 Seventh International Conference on the Quality of Information and Communications Technology*. 106–115. https://doi.org/10.1109/QUATIC.2010.16

[5] Zhifei Chen, Lin Chen, Wanwangying Ma, Xiaoyu Zhou, Yuming Zhou, and Baowen Xu. 2018. Understanding metric-based detectable smells in Python software: A comparative study. *Inf. Softw. Technol.* 94 (2018), 14–29. https://doi.org/10.1016/j.infsof.2017.09.011

[6] Ananta Kumar Das, Shikhar Yadav, and Subhasish Dhal. 2019. Detecting code smells using deep learning. In *TENCON 2019-2019 IEEE Region 10 Conference (TENCON)*. IEEE, 2081–2086.

[7] Amin Milani Fard and Ali Mesbah. 2013. JSNOSE: Detecting JavaScript Code Smells. In *13th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2013, Eindhoven, Netherlands, September 22-23, 2013*. IEEE Computer Society, 116–125. https://doi.org/10.1109/SCAM.2013.6648192

[8] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., USA.

[9] Shizhe Fu and Beijun Shen. 2015. Code Bad Smell Detection through Evolutionary Data Mining. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2015, Beijing, China, October 22-23, 2015*. IEEE Computer Society, 41–49. https://doi.org/10.1109/ESEM.2015.7321194

[10] Mouna Hadj-Kacem and Nadia Bouassida. 2018. A Hybrid Approach To Detect Code Smells using Deep Learning.. In *ENASE*. 137–146.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[12] Yaoshiang Ho and Samuel Wookey. 2019. The real-world-weight cross-entropy loss function: Modeling the costs of mislabeling. *IEEE access* 8 (2019), 4806–4813.

[13] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*. pmlr, 448–456.

[14] Amandeep Kaur, Sushma Jain, and Shivani Goel. 2017. A support vector machine based approach for code smell detection. In *2017 International Conference on Machine Learning and Data Science (MLDS)*. IEEE, 9–14.

[15] Wael Kessentini, Marouane Kessentini, Houari Sahraoui, Slim Bechikh, and Ali Ouni. 2014. A cooperative parallel search-based software engineering approach for code-smells detection. *IEEE Transactions on Software Engineering* 40, 9 (2014), 841–861.

[16] Foutse Khomh, Stephane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. 2011. BDTEX: A GQM-based Bayesian approach for the detection of antipatterns. *Journal of Systems and Software* 84, 4 (2011), 559–572.

[17] Michele Lanza and Radu Marinescu. 2006. *Object-Oriented Metrics in Practice - Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer. https://doi.org/10.1007/3-540-39538-5

[18] Hui Liu, Jiahao Jin, Zhifeng Xu, Yanzhen Zou, Yifan Bu, and Lu Zhang. 2019. Deep learning based code smell detection. *IEEE transactions on Software Engineering* 47, 9 (2019), 1811–1837.

[19] Radu Marinescu. 2004. Detection Strategies: Metrics-Based Rules for Detecting Design Flaws. In *20th International Conference on Software Maintenance (ICSM 2004), 11-17 September 2004, Chicago, IL, USA*. IEEE Computer Society, 350–359. https://doi.org/10.1109/ICSM.2004.1357820

[20] Radu Marinescu. 2005. Measurement and Quality in Object-Oriented Design. In *21st IEEE International Conference on Software Maintenance (ICSM 2005), 25-30 September 2005, Budapest, Hungary*. IEEE Computer Society, 701–704. https://doi.org/10.1109/ICSM.2005.63

[21] Tom Mens and Tom Tourwé. 2004. A survey of software refactoring. *IEEE Transactions on software engineering* 30, 2 (2004), 126–139.

[22] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur. 2010. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Trans. Software Eng.* 36, 1 (2010), 20–36. https://doi.org/10.1109/TSE.2009.50

[23] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. 2015. Mining Version Histories for Detecting Code Smells. *IEEE Trans. Software Eng.* 41, 5 (2015), 462–489. https://doi.org/10.1109/TSE.2014.2372760

[24] Tushar Sharma, Vasiliki Efstathiou, Panos Louridas, and Diomidis Spinellis. 2021. Code smell detection by deep direct-learning and transfer-learning. *Journal of Systems and Software* 176 (2021), 110936.

[25] Tushar Sharma, Pratibha Mishra, and Rohit Tiwari. 2016. Designite: a software design quality assessment tool. In *Proceedings of the 1st International Workshop on Bringing Architectural Design Thinking into Developers' Daily Activities, BRIDGE@ICSE 2016, Austin, Texas, USA, May 17, 2016*. ACM, 1–4. https://doi.org/10.1145/2896935.2896938

[26] Tushar Sharma and Diomidis Spinellis. 2018. A survey on software smells. *Journal of Systems and Software* 138 (2018), 158–173.

[27] Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. 2014. *Refactoring for Software Design Smells: Managing Technical Debt* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[28] Santiago A. Vidal, Claudia A. Marcos, and Jorge Andrés Díaz Pace. 2016. An approach to prioritize code smells for refactoring. *Autom. Softw. Eng.* 23, 3 (2016), 501–532. https://doi.org/10.1007/s10515-014-0175-x

[29] Yang Zhang, Chuyan Ge, Shuai Hong, Ruili Tian, Chunhao Dong, and Jingjing Liu. 2022. DeleSmell: Code smell detection based on deep learning and latent semantic analysis. *Knowledge-Based Systems* 255 (2022), 109737.