Multi-Granularity Code Smell Detection using Deep Learning Method based on Abstract Syntax Tree

Weiwei Xu, Xiaofang Zhang
School of Computer Science and Technology
Soochow University
Suzhou, China
Email: xfzhang@suda.edu.cn

***Abstract*—Code smell refers to poor design that is perceived to have a negative impact on readability and maintainability during software evolution, and it implies the possibility of refactoring. Therefore, the effective detection of code smell is of great importance. Many approaches including metric-based, heuristic-based, and machine learning approaches have been proposed to detect code smells. However, all these methods use manually selected features, which is highly subjective and difficult to select the most appropriate features. Recently, deep learning methods without extensive feature engineering have been proposed. Nevertheless, these token-based approaches may not achieve good results because they ignore many semantic and structural information of source code. To this end, we propose a novel deep learning approach based on abstract syntax trees(ASTs) to detect multi-granularity code smells, which captures the semantic and structural features of code fragments from the ASTs. The experimental results on four types of smells show that this approach achieves better results than the state-of-the-art approaches for detecting code smells with different granularities.**

***Index Terms*—code smell, abstract syntax tree, deep learning**

I. Introduction

Code smell refers to some bad designs in the code, which often has a bad effect on the readability and maintainability of the software. Furthermore, code smell suggests the possibility of refactoring [1], so detecting code smells in a timely and effective manner can be a guide for developers in refactoring. Software engineering researchers have done a lot of research on the definition, causes, and effects of code smell [2].

A number of approaches have been proposed to detect different types of code smells in source code. Metric-based [3] and heuristic-based [4] approaches are the traditional ways to detect code smells. However, most of them have strong limitations because they all rely on manually designed heuristics to obtain final results from manually selected features. Picking the most appropriate features and building heuristics are very difficult, and computing the corresponding metrics for the target source code is a considerable amount of work. In recent years, many scholars have proposed to use machine learning methods such as Support Vector Machine, Naive Bayes and Logistic Regression to detect code smells. Although machine learning methods avoid manually designed heuristics [5], existing machine learning methods for detecting code smells are still in need of further research and improvement [6]. Machine learning methods require a collection of features extracted from the source code, i.e., they still require external tools to compute many metrics of source code.

Recently, Sharma et al. experimented with a deep learning approach without extensive feature engineering to detect code smells and verified the feasibility of the approach on several smells [7]. Deep learning models can learn intrinsic features during training to classify samples, but existing deep learning methods have some limitations as follows.

- The deep learning models are token-based. The token-based code representation may lose the rich semantic and structural information in the source code.
- Existing methods focus only on code smells with small granularity, lack of experimentation on code smells with larger granularity.
- A universally well-performing deep learning model was not found for different code smells.

To address these problems, we propose a novel abstract syntax trees(ASTs) based code smell detection approach(AST-CSD). The approach extracts the ASTs from the code fragments and forms sequences of statement trees by splitting the complete AST into several subtrees. First, we encode the sequences of statement trees and then extract semantic and structural features from the sequences using bi-directional GRU [8] and maximum pooling. Final vector representations of code fragments can be obtained after that. At last, the final detection result is derived through several fully-connected layers. We apply the AST-based approach to 500 high-quality Java projects from GitHub. Better results are achieved than the state-of-the-art deep learning models, not only on one type of small-grained code smells but also on three types of larger-grained code smells.

The main contributions of this paper are as follows.

- We propose a deep learning approach based on ASTs to detect code smells. To the best of our knowledge, we are the first to conduct research on code smell detection using deep learning methods based on ASTs.
- In addition to smells with small granularity, we focus on detection of code smells with larger granularity, bridging

DOI reference number: 10.18293/SEKE2021-014

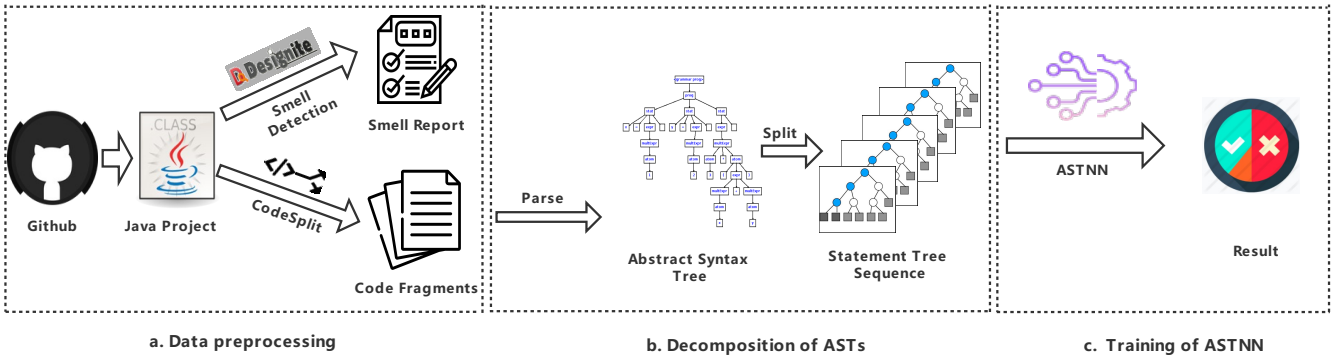


Fig. 1. Overview of AST-CSD

the gap of previous work and enabling detection of multi-granularity code smells.
- For different code smells, we conduct extensive experiments to find out the parameter configuration that makes the model perform best.

The rest of this paper is organized as follows. Section II introduces the background; Our AST-CSD approach is introduced in Section III; Section IV describes the experimental setup and results are in Section V; The conclusion of this paper and the future work are presented in Section VI.

II. Background

A. Code Smell

Fowler and Beck first introduced the notion of code smell [1] and defined it as *"certain structures in the code that suggest (or sometimes scream) for refactoring."* Code smell affects the readability and maintainability of programs and has an impact on the software development and evolution process.

Code smells can be divided into implementation [1], design [9] and architecture [10] smells in the order of size according to their granularity or scope [7]. Implementation Smells have the smallest granularity and scope, and they usually occur on methods. Design Smells, which are in the middle granularity, typically occur at the class level. Architecture smells have the greatest granularity, often involving multiple components, and their impact is at the system level.

B. Abstract Syntax Tree

An abstract syntax tree (AST) is a tree representation of the abstract syntactic structure of source code written in a programming language [11]. The abstract syntax tree clearly describes the structure of the source code. In many existing studies, source code is parsed into abstract syntax trees to produce code representations that capture the semantic relationships between different code elements [12], [13]. Code representation based on abstract syntax trees is now being used for code clone detection [11], defect prediction [14], auto program repair [15], and other problems. In metric-based code smell detection methods, abstract syntax trees may also be used to compute a set of source code metrics [7], [16]. However, these methods do not take advantage of the rich semantic and structural information in the abstract syntax trees.

C. Motivation

Existing deep learning methods for code smell detection are token-based. The token-based code representation approaches treat code fragments as natural language texts. Although code fragments have some similarity with natural language texts, code fragments should not be treated simply as natural language texts because there is rich structural information in code fragments [17]. For example, two statements located closely to each other, one outside the loop body and one inside the loop body, are semantically disjoint. But the token-based approach does not reflect this disjoint relationship well.

Recent work has demonstrated the superiority of an AST-based approach to code representation over a token-based approach [17], [18]. Intuitively, the rich semantic and structural information in AST will help us in smell detection. For example, when we detect the code smells such as *complex method*, there are three adjacent loop statements in the method, and the token-based method does not clearly show whether the three loop statements are nested or not. By contrast in AST, we can determine by observing whether the three statements are at the same depth of the tree. Whether the loop statements are nested or not obviously is critical to judge the complexity of the method. Therefore, we believe that more semantic and structural information in the AST-based code representation approach is of great help in code smell detection.

III. Approach

This section introduces the method we use to detect code smells. Figure 1 gives an overview of our method.

A. Data preprocessing

We first use the CodeSplit[1] to split all the projects downloaded from Github into class-level and method-level code fragments. Then we use Designite [19] to find out the smells contained in the source code and generate smell reports. Based on the smell report, we divide the code fragments that have

[1]https://github.com/tushartushar/CodeSplitJava

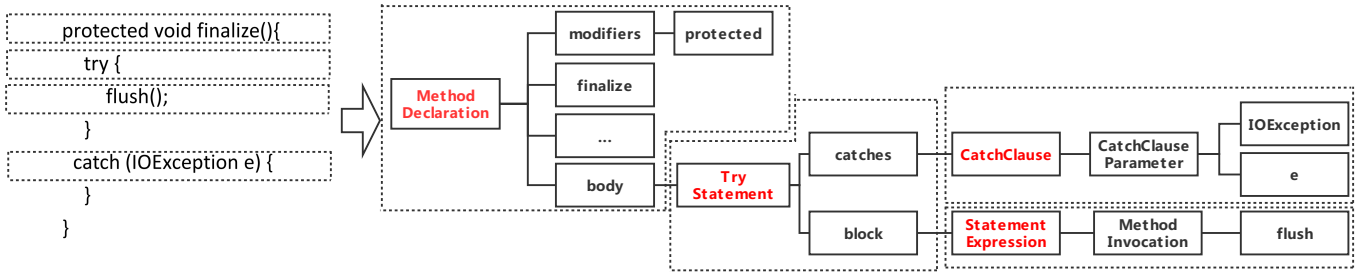


Fig. 2. The process of splitting a complete abstract syntax tree into statement trees

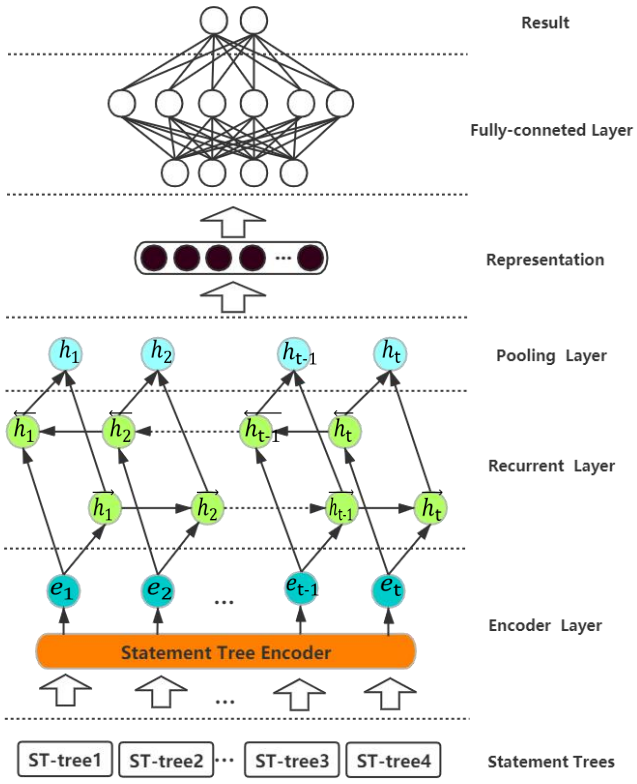


Fig. 3. The structure of ASTNN

been split up to corresponding granularity into two categories, one containing smells and one without.

B. Decomposition of ASTs

We use Javalang[2] to parse the code fragment and get the AST of it, and then we store the AST with its corresponding label. Figure 2 shows the decomposition of an AST, the left side of the figure shows the code fragment of a method, and the right side shows its complete abstract syntax tree. According to the method of Jian Zhang et al. [17], we split the each statement like *Try* statement into two parts which has a header and a body containing a lot of statements. The statement trees are marked with dashed lines in Figure 2, and the red node is the root node of the statement tree. By preorder traversal, we obtain a sequence of statement trees. We store the sequences of statement trees and their corresponding labels of whether they contain the smell, which are later used to train the ASTNN model.

[2]https://github.com/c2nes/javalang

C. Training of ASTNN

We use the ASTNN model proposed in [17] and Figure 3 shows the structure of it. The model includes three parts: encoding statement trees, representing statement sequences and classification.

1) Encoding Statement Trees: To obtain vector representations of statements, we use an RvNN-based statement encoder. There are many syntactic symbols in ASTs, and we obtain all the symbols in ASTs as a corpus by traversing ASTs in preorder. Then we use the word2vec [20] to learn unsupervised vectors of the symbols. Given a statement tree t, let n denote a non-leaf node and let C denote the number of its children nodes. In the beginning, the lexical vector of node n can be obtained by:

$$v_n = {W_e}^{\top} x_n \tag{1}$$

where $W_e \in \mathbb{R}^{|V| \times d}$ is the pre-trained embedding parameters with the vocabulary size V and the embedding dimension of symbols d, v_n is the embedding of symbol n and x_n is its one-hot representation. Then the vector representation of node n can be calculated using the following equation:

$$h = \sigma({W_n}^{\top} v_n + \sum_{i \in [1,C]} h_i + b_n) \tag{2}$$

where $W_n \in \mathbb{R}^{d \times k}$ is the weight matrix and k is the encoding dimension, h_i is the hidden state of its each child, b_n is a bias term, σ is the activation function, for which we use identify function in the method, and h is the latest hidden state. We can recursively compute the vector representations of all nodes in the statement tree t. Finally, we obtain the vector representation of the entire statement tree t by maximum pooling sampling:

$$e_t = [max(h_{i1}), max(h_{i2}), ..., max(h_{ik})], i = 1, ...N \tag{3}$$

where N is the number of nodes in t.

2) Representing the Sequence of Statement Trees: In the previous procedure, we can get vector representations of all statement trees, so for each AST, we have a sequence of statement tree vectors. Using this sequence of statement tree vectors, we then use bi-directional GRU [8] to capture the naturalness of statements. Finally, we sample the most important features of these states by means of the max pooling. At this point, we obtain a vector representation of the code fragment.

3) Classification: After obtaining the vector representations of the code fragments, we feed them into a neural network consisting of several fully connected layers and classify them into two classes, one containing smells and one without.

IV. Experimental settings

A. Projects and datasets

We choose to use the same dataset, 500 high-quality Java projects covering a variety of functions from Github, as used in [7]. Since implementation smells and design smells occur at the method level and class level, respectively, and a class usually contains many methods, if the same number of projects are used for both types of smells in the experiment, the former will have a much larger sample size. Consequently, for the implementation smells with small granularity(i.e., method level), we select 100 projects randomly from 500 projects, while for smells with large granularity, we use all 500 projects. For samples with different granularity, we process them separately: removing duplicate samples and deleting overlong samples with the length over one standard deviation away from the mean. The goal of this procedure is to keep the training set within a reasonable range and avoid wasting memory and processing resources.

We divide all samples into three parts, 70% as the training set, 10% as the validation set, and 20% as the test set. To reduce the impact of the extreme imbalance, we balance the positive and negative samples in the training set. The number of both positive and negative samples in the training set is limited to 5000, and if there are more negative samples than positive samples, the number of negative samples is reduced to the same as the positive samples. Table I shows the number of positive and negative samples used in our experiment.

TABLE I
NUMBER OF POSITIVE(P) AND NEGATIVE(N) SAMPLES

	Training Set		Validation Set		Test set	
Smell	P	N	P	N	P	N
Insufficient Modularization(IM)	5000	5000	927	14170	1857	27341
Deficient Encapsulation(DE)	5000	5000	1824	13273	3651	26547
Feature Envy(FE)	1230	1230	175	14922	353	29845
Empty Catch Block(ECB)	359	359	51	4708	103	9418

B. Selection of code smells

To further explore the effectiveness of deep learning methods in detecting smells with different granularity, in addition to implementation smells, our experiments focus on the design smells with larger granularity which are more difficult to detect.

We choose *Insufficient Modularization* (IM, i.e., the class has not been completely decomposed), *Deficient Encapsulation* (DE, i.e., the declared accessibility of one or more members of the class is more permissive than actually required), *Feature Envy* (FE, i.e., the class has a method that uses methods and data of other classes more than using its own ones and seems more interested in a class other than the one it actually is in). We select these design smells because they are representative due to their high frequency of occurrence in the 500 projects.

What's more, we choose *Empty Catch Block* (ECB, i.e., a catch block of an exception is empty), which is an implementation smell. This smell was also chosen in the experiments of Sharma et al. [7], and we use it to verify that our model also has a good performance on small-grained smells.

TABLE II
VALUES OF HYPER-PARAMETERS FOR CNN MODELS

Hyper-parameter	Values
Number of repetitions of the set of hidden unit(N)	{1, 2, 3}
Filters in convolution layer(F)	{8, 16, 32, 64}
Kernel size in convolution layer(K)	{5, 7, 11}
Pooling window size in max pooling layer(W)	{2, 3, 4, 5}

TABLE III
VALUES OF HYPER-PARAMETERS FOR RNN MODEL

Hyper-parameter	Values
Number of repetitions of the set of hidden unit(N)	{1, 2, 3}
Embedding dimensions(E)	{16, 32}
LSTM units(U)	{32, 64, 128}

C. Baseline setting

In this paper, we select the following three baseline methods proposed by Sharma et al. [7] as comparative methods to estimate the performance of our proposed method:

1) CNN-1D Model: In this model, each input instance is represented by a 1D array of tokens. The model extract features through convolution, batch normalization, and max pooling layers. Finally, the fully-connected layers are used to make predictions about whether a given instance belongs to the positive or negative class.

2) CNN-2D Model: The CNN-2D model is similar to the CNN-1D model, except that each input instance of CNN-2D model is a 2D array of tokens, which delineates the source code statement by statement.

3) RNN Model: The RNN model has the same input as CNN-1D, but unlike CNN-1D, RNN captures features using an embedding layer and a LSTM layer.

We obtain the hyper-parameters configurations for the baseline methods according to [7]. Tables II and III show the values of the hyper-parameters for the CNN and RNN models. All combinations of hyper-parameters are performed to confirm the best configuration of baseline methods.

TABLE IV
VALUES OF HYPER-PARAMETERS FOR AST-CSD

Hyper-parameter	Values
Number of fully-connected layers(FC)	{1, 2, 3, 4}
Embedding and encoding dimensions(EE)	{64, 128, 256, 512}
Dimensions of hidden states in GRU(H)	{50, 75, 100, 125}

TABLE V
PERFORMANCE OF AST-CSD ON THE OPTIMAL CONFIGURATION

	Performance			Configuration		
Smells	P	R	F-measure	FC	EE	H
IM	0.65	0.92	0.76	2	64	75
DE	0.95	0.95	0.95	2	128	125
FE	0.11	0.53	0.17	2	256	75
ECB	0.26	0.84	0.40	4	512	50

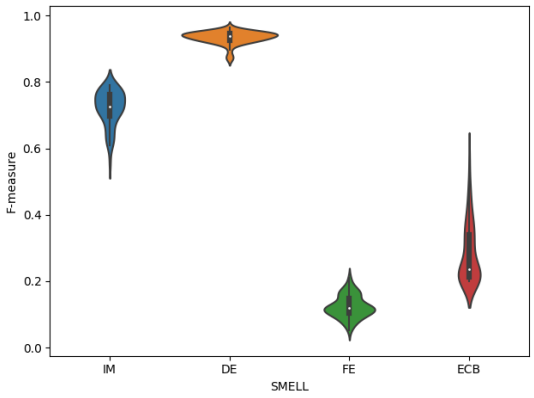


Fig. 4. Violin plot of F-measure exhibit by AST-CSD

D. Evaluation

Due to the extremely unbalanced distribution of positive and negative samples in real projects, we avoid comparing the $accuracy$ of each model because if a model predicts all samples as negative, it will still have high accuracy. We choose $precision$, $recall$ and $F-measure$ as the evaluation metrics. They are difined as follows:

$$precision = \frac{True\ Positive}{True\ Positive + False\ Positive} \tag{4}$$

$$recall = \frac{True\ Positive}{True\ Positive + False\ Negative} \tag{5}$$

$$F - measure = \frac{2 \times precision \times recall}{precision + recall} \tag{6}$$

V. EXPERIMENTAL RESULTS

In this section, we mainly focus on answering the following research questions:

RQ1: How does the AST-based approach perform under different configurations for multi-granularity code smells?

RQ2: Is the AST-based approach better than the token-based approaches in detecting code smells with different granularities?

RQ3: Is the AST-based approach significantly superior?

A. RQ1:How does the AST-based approach perform under different configurations for multi-granularity code smells?

Table IV shows the values of the different hyper-parameters for our approach. We perform 64 combinations of hyper-parameters to get the best configuration of our approach.

Table V lists the performance of our AST-based approach AST-CSD on the optimal configuration. From the table, we can easily see that the AST-based approach does not perform equally on the four types of smells, and the combination of hyper-parameters that make the approach perform optimally for different smells also varies.

Figure 4 shows the violin plot of performance of the approach under all configurations for four smells. Among the four smells, the AST-based method has good performance in detecting *Deficient Encapsulation* and *Insufficient Modularization* smell. However, this AST-based method did not perform very well in detecting *Feature Envy* smell. We believe this is somewhat related to the extremely unbalanced ratio of positive and negative samples for this smell, which reaches a ratio of 353:29845 in the test set. For *Empty Catch Block* smell, which is an implementation smell, it can be seen from Figure 3 that different hyper-parameters have a greater impact on the performance.

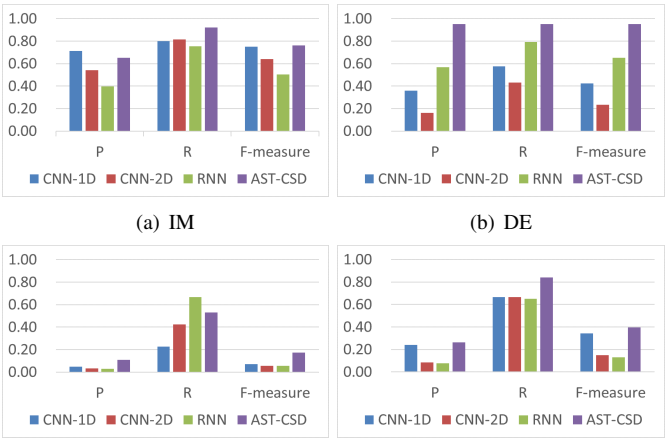
B. RQ2: Is the AST-based approach better than the token-based approaches in detecting code smells with different granularities?

We first perform parameter search for the baseline methods to find the combination of hyper-parameters that has the best performance. Table VI shows the hyper-parameters of the three token-based methods when they achieve the best performance.

TABLE VI
PERFORMANCE OF THE TOKEN-BASED APPROACHES ON THE OPTIMAL CONFIGURATION

		Performance			Configuration					
	Smells	P	R	F-measure	N	F	K	W	E	U
CNN-1D	IM	0.71	0.80	0.75	3	64	7	4	-	-
	DE	0.36	0.57	0.43	3	32	5	4	-	-
	FE	0.05	0.23	0.07	3	64	7	4	-	-
	ECB	0.24	0.66	0.34	3	32	5	4	-	-
CNN-2D	IM	0.54	0.81	0.64	2	8	5	2	-	-
	DE	0.16	0.43	0.23	3	16	5	4	-	-
	FE	0.03	0.43	0.06	3	16	7	2	-	-
	ECB	0.09	0.66	0.15	2	64	7	2	-	-
RNN	IM	0.40	0.76	0.50	1	-	-	-	32	64
	DE	0.57	0.79	0.65	2	-	-	-	32	32
	FE	0.03	0.67	0.05	3	-	-	-	32	128
	ECB	0.08	0.65	0.13	3	-	-	-	16	32

After getting the optimal configurations of each method, we repeat the training and testing of each approach on the optimal configuration for 30 times. Figure 5 shows the average performance of the AST-based approach compared to the token-based approaches on the four smells. As shown in Figure 5,



(a) IM (b) DE

(c) FE (d) ECB

Fig. 5. The performance of token-based and AST-based approaches on four code smells

the AST-based approach achieves better results than the token-based approaches on all three smells with large granularity. In addition, the AST-based approach also obtains better results for smell *Empty Catch Block* with small granularity.

C. RQ3: Is the AST-based approach significantly superior?

To further analyze the performance of AST-based approach and baseline approaches, Wilcoxon signed-rank test and Cliff's delta test are conducted. If $p-value$ of Wilcoxon signed-rank test is less than 0.05, the two matched samples are significantly different. Cliff's delta test can be used as a complementary analysis to Wilcoxon signed-rank test, and Cliff's delta test can measure the effective level of difference between the two sets of observation data. Table VII shows Cliff's delta values($|\delta|$) and the corresponding effective levels.

We use the Win/Tie/Loss indicator to compare the performance of different methods. Specifically, if the AST-based method outperforms a baseline method with the $p-value$ of Wilcoxon signed-rank test less than 0.05 and the Cliff's delta value greater than or equal to 0.147, the difference between these two methods is statistical significant, in which case we consider AST-CSD to win. Conversely, if the baseline model is better than the AST-based method and the difference between them is significant, we consider AST-CSD to lose. In other cases, we consider them to be tied. What's more, '+' or '-' before the effective level is to represent the positive or negative Cliff's delta values. '+' means the AST-based method is superior.

As shown in Table VIII, our AST-based method significantly outperforms other token-based methods in detecting all four smells with different granularities.

VI. Threats to Validity

A. Internal validity

One of the main factors affecting internal validity is the experimental environment. We use the Designite tool to detect smells, which is used to generate labels for the training data, and view its results as ground truth. Since it is widely used in related work [7], [19], we think it is reliable to use the tool to detect code smells. In addition, the code is carefully reviewed and tested to ensure that the code we used to build the model was error-free.

TABLE VII
Mappings between Cliff's delta values and their effective levels

Cliff's delta	Effective levels
$\|\delta\| < 0.147$	Negligible
$0.147 \leq \|\delta\| < 0.33$	Small
$0.33 \leq \|\delta\| < 0.474$	Medium
$0.474 \leq \|\delta\|$	Large

TABLE VIII
Win/Tie/Loss indicators on $F-measure$ values of token-based methods and AST-CSD

Smells	AST-CSD vs CNN-1D	AST-CSD vs CNN-2D	AST-CSD vs RNN
IM	<0.05(+Medium)	<0.05(+Large)	<0.05(+Large)
DE	<0.05(+Large)	<0.05(+Large)	<0.05(+Large)
FE	<0.05(+Large)	<0.05(+Large)	<0.05(+Large)
ECB	<0.05(+Medium)	<0.05(+Large)	<0.05(+Large)
Win/Tie/Loss	4/0/0	4/0/0	4/0/0

B. External validity

External validity refers to the validity of generalization of research results. In this study, we use 500 open source Java projects. Experiments on other datasets (non-Java projects or industrial projects) will help to further validate our approach.

VII. Conclusion and Future Work

In this paper, we propose a new deep learning method based on ASTs to detect code smells. We exploit the rich semantic and structural information in the AST to generate the final feature representations of code fragments. Experiments on smells with different granularities show that our method is significantly better than state-of-the-art deep learning methods in terms of $F-measure$.

As future work, smells with greater granularity, i. e., architectural smells, need to be considered. When using deep learning methods to detect smells with greater granularity, how to use the numerous components involved in architectural smells as input is a question worth investigating. What's more, it is of great value to extend our approach to other programming languages, such as Python and C++.

Acknowledgment

This work is partially supported by the National Natural Science Foundation of China(61772263, 61772014, 61872177), Collaborative Innovation Center of Novel Software Technology and Industrialization, Undergraduate Training Program for Innovation and Entrepreneurship of Soochow University(202010285141H), and the Priority Academic Program Development of Jiangsu Higher Education Institutions.

REFERENCES

[1] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[2] T. Sharma and D. Spinellis, "A survey on software smells," *Journal of Systems and Software*, vol. 138, pp. 158–173, 2018.

[3] M. Salehie, S. Li, and L. Tahvildari, "A metric-based heuristic framework to detect object-oriented design flaws," in *14th IEEE International Conference on Program Comprehension (ICPC'06)*. IEEE, 2006, pp. 159–168.

[4] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2009.

[5] H. Liu, J. Jin, Z. Xu, Y. Bu, Y. Zou, and L. Zhang, "Deep learning based code smell detection," *IEEE transactions on Software Engineering(Early Access)*, 2019.

[6] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, "Detecting code smells using machine learning techniques: are we there yet?" in *2018 ieee 25th international conference on software analysis, evolution and reengineering*. IEEE, 2018, pp. 612–621.

[7] T. Sharma, V. Efstathiou, P. Louridas, and D. Spinellis, "On the feasibility of transfer-learning code smells using deep learning," *arXiv preprint arXiv:1904.03031*, 2019.

[8] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[9] G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for software design smells: managing technical debt*. Morgan Kaufmann, 2014.

[10] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Identifying architectural bad smells," in *2009 13th European Conference on Software Maintenance and Reengineering*. IEEE, 2009, pp. 255–258.

[11] C. Fang, Z. Liu, Y. Shi, J. Huang, and Q. Shi, "Functional code clone detection with syntax and semantics fusion learning," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 516–527.

[12] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[13] H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code." in *IJCAI*, 2017, pp. 3034–3040.

[14] H. K. Dam, T. Pham, S. W. Ng, T. Tran, J. Grundy, A. Ghose, T. Kim, and C.-J. Kim, "A deep tree-based model for software defect prediction," *arXiv preprint arXiv:1802.00921*, 2018.

[15] Y. Li, S. Wang, and T. N. Nguyen, "Improving automated program repair using two-layer tree-based neural networks," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, 2020, pp. 316–317.

[16] P. Singh, S. Singh, and J. Kaur, "Tool for generating code metrics for c# source code using abstract syntax tree technique," *ACM SIGSOFT Software Engineering Notes*, vol. 38, no. 5, pp. 1–6, 2013.

[17] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 783–794.

[18] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 30, no. 1, 2016.

[19] T. Sharma, P. Mishra, and R. Tiwari, "Designite: A software design quality assessment tool," in *Proceedings of the 1st International Workshop on Bringing Architectural Design Thinking into Developers' Daily Activities*, 2016, pp. 1–4.

[20] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *arXiv preprint arXiv:1310.4546*, 2013.