Multi-Label Code Smell Detection with Hybrid Model based on Deep Learning

Yichen Li
School of Computer Science and Technology
Soochow University
Suzhou, China
Email: ycli1024@stu.suda.edu.cn

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

***Abstract*—Code smell is an indicator of potential problems in a software design that have a negative impact on readability and maintainability. Hence, it is essential for developers to make out the code smell to get tips on code maintenance in time. Fortunately, many approaches like metric-based, heuristic-based, machine-learning based and deep-learning based have been proposed to detect code smells. However, existing methods, using the simple code representation to describe different code smells unilaterally, cannot efficiently extract enough rich information from source code. What is more, one code snippet often has several code smells at the same time and there is a lack of multi-label code smell detection based on deep learning. In this paper, we propose a hybrid model with multi-level code representation to further optimize the code smell detection. First, we parse the code into the abstract syntax tree(AST) with control and data flow edges and the graph convolution network is applied to get the prediction at the syntactic and semantic level. Then we use the bidirectional long-short term memory network with attention mechanism to analyze the code tokens at the token-level in the meanwhile. Finally we get the fusion prediction result of the models. Experimental results show that our model can perform outstanding not only in single code smell detection but also in multi-label code smell detection.**

***Index Terms*—Code smell, multi-label, code representation, hybrid model, deep learning**

I. Introduction

Code Smells indicate problems related to aspects of code quality such as understandability and modifiability, and imply the possibility of refactoring [1]. So Code smell analysis, which allows people to integrate both assessment and improvement into the software evolution process, is of great importance. Software engineering researchers have studied the concept in detail and explored various aspects associated with code smells, including causes, impacts, and detection methods [2].

Many approaches have been proposed to detect code smells. Traditionally, metric-based [3] and heuristic-based methods [4] use the manually designed regulations to extract the features inside the code. However, it's difficult for developers to reach an agreement on the appropriate rules and corresponding metrics. Machine-learning based methods [5], which apply Support Vector Machine, Naive Bayes and Logistic Regression, still have a long way to go to conquer problems of manually selected features and extra computation tools [6]. In recent years, a universally well-performing deep learning model [7] has been applied to code smell detection. In addition, the abstract syntax tree(AST) has been used to extract the syntactic features from the source code to detect the code smell [8].

Furthermore, multi-label code smell detection has attracted attention. Since the code snippet tends to have many code smells that may lead to potential problems, multi-label code smell detection means to find out all code smells inside the code snippet instead of one at a time. Guhhulothu et al. carried the experiment on a multi-label dataset of combining labels of two code smell datasets and Random-Forest was applied to detect two code smells at the same time [9]. All of the methods above solve the problem to some extent, but they all have the limitations below:

- The models just use code tokens or ASTs simply. Such methods will lose part of the information that helps recognize each code smell more efficiently.
- No one has proposed a model which can make the multi-label classification based on deep learning. Since the code snippet may has several code smells at the same time, it's necessary to propose an efficient and convenient model to find out code smells.

To address these limitations, in this paper we propose a **h**ybrid **m**odel with **m**ulti-**l**evel code representation(HMML). We first parse the AST from the source code and add the control and data flow edges [10] to get the code property graph. Then we apply the graph convolution network(GCN) [11] to learn information from the high dimensions at the syntactic and semantic level. Meanwhile, we use the bidirectional long-short term memory(LSTM) network with attention to analyze the code tokens at the token-level. Finally, we use the outputs of two models by weight to get the predication result. What is more, all of the models mentioned in this paper have been optimized to fit for the multi-label classification task. We apply our HMML method to 100 high-quality Java projects from Github. Better results have been achieved not only on multi-label code smell detection but on some single code smell detections.

The main contributions of this article are as follows:

- We propose a hybrid model that extracts the multi-level

DOI reference number: 10.18293/SEKE2022-077

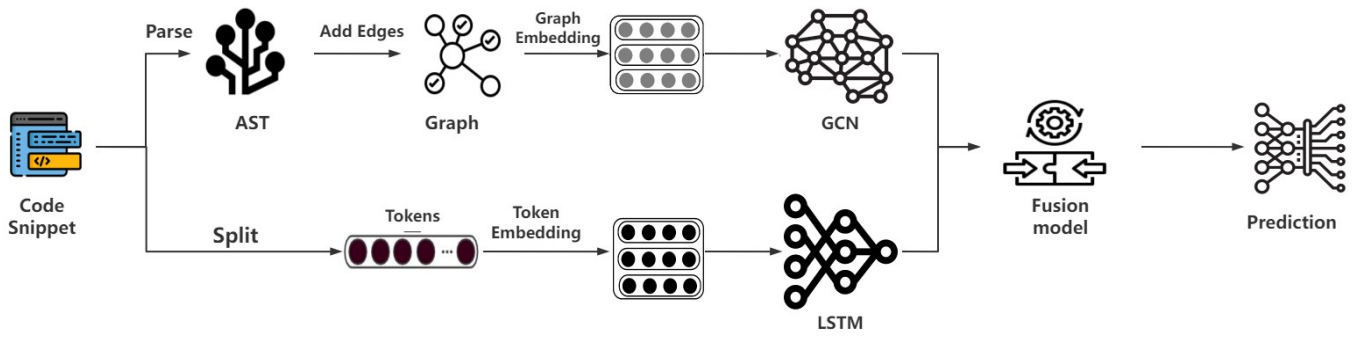

Fig. 1. Overview of the HMML

code representation information and separately applies the appropriate deep learning neural network.
- We are the first to carry out the multi-label code smell detection based on the deep learning method and achieve a good result.
- We modify many other approaches to fit into multi-label classification tasks and conduct extensive experiments to find the maximum capacity and best configuration.

The rest of this paper is organized as follows. Section II introduces the background; Our HMML method 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

Code smells were first introduced by Fowler [1] as "structures with technical debt which affect maintainability negatively". Code smells imply the possibility of refactoring and have an impact on software development and evaluation. Fowler categorized code smells as implementation, design [12] and architecture [13] smells based on the scope and granularity [14].

B. Abstract syntax tree

Abstract Syntax Tree (AST) is a tree representation of the abstract syntactic structure of source code written in a programming language [15]. Developers can get the declaration statements, assignment statements, operation statements and realize operations by analyzing the tree structures [16].Nowadays, Some studies use AST-based approaches for source code clone detection [15], program translation [17], and code smell detection [8].

C. Motivation

Existing methods take a one-sided approach to the code smell detection problem. On the one hand, no one has applied the state-of-art deep learning to the multi-label code smell detection. On the other hand, many researchers focus on the token-based method [7] or AST-based method [8]. Although code fragments have some similarities with natural language texts and AST extracts some syntactic information, the information is still far from enough. Some code smells are caused by several aspects and the simple code representation fails to distinguish them. For example, *Long Method* is a general code smell and it is caused by the length of the code, long comment, complex conditional statement and messy loop. Existing methods cannot catch the cause of the code smell accurately because token-based methods ignore the syntax information by treating each code seperately and AST-based methods lose the words meaning and information about the comment, code length when compiling the code.

In the meanwhile, recent work has demonstrated the superiority of a graph-based approach to code representation over other approach [10]. Intuitively, the rich semantic and structural information in the graph will help us in smell detection. In terms of the *Missing default*, AST-based methods simply treat the statement as branch of the tree and ignore the possible logical errors linked to the data flow due to the missing default. By contrast, the graph-based methods with control and flow data can vividly show the change by adding extra edges among statements. To ensure the model's ability to catch different code smells, we fuse the token-based approach and graph-based approach to entirely get the structural, syntactic, semantic information to detect code smells.

III. APPROACH

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

To extract tokens and AST from Java programs, we use a python package javalang[1]. We use the method proposed in [10] to add the control and data flow edges. We focus on the following essential control flow types: *Sequential execution*, *Case statements*, *While* and *For loops*, which are linked to code smells mentioned in our motivation. In this paper, we use two different neural networks: a traditional LSTM for word tokens and a GCN that catches the information inside the graph.

1) LSTM Model: We use bidirectional long-short term memory network with attention mechanism to capture the

[1]https://github.com/c2snet/javalang

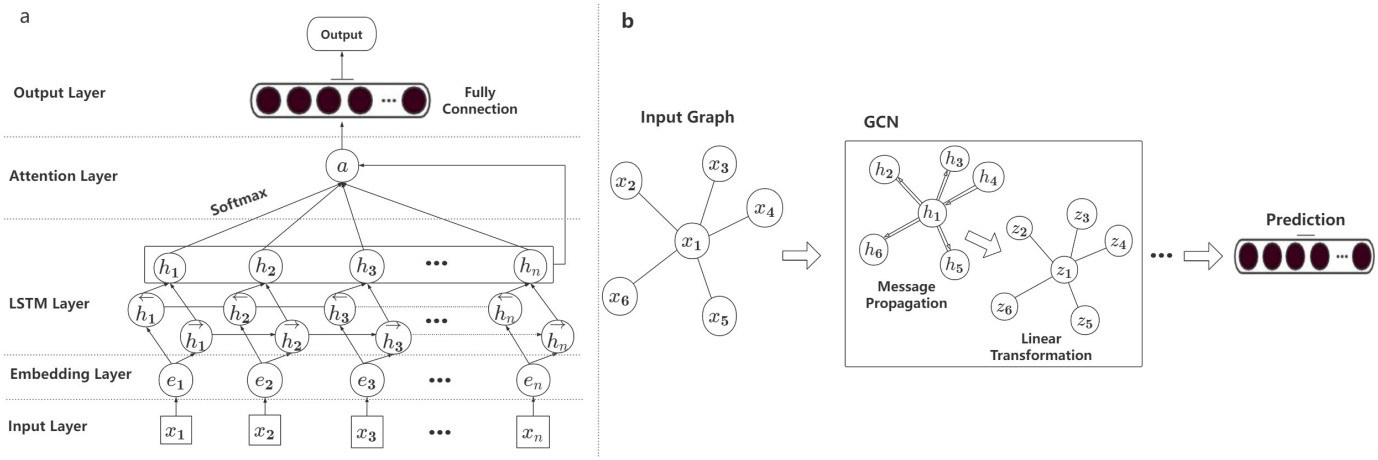

Fig. 2. Details of the Model

information in front and behind of the current position. Figure 2(a) shows details of the LSTM Model.

The attention is designed to selectively focus on parts of the source sentence during translation. We use global attention in this model to extract source context vector.

$$c_j = \sum_{i=1}^{|x|} a_{ij} h_i \tag{1}$$

where a_{ij} is the attention weights of hidden state h_i. The attention mechanism will give more weight to the hidden state vectors of important tokens.

$$r_{ij} = h_i * c_j \tag{2}$$

$$y = Sigmoid(W_s r_{ij} + b_s) \tag{3}$$

where W_s, b_s are parameters for Sigmoid layer. Here we use the Sigmoid layer as output layer to reveal the multi-label classification task.

2) GCN Model: Figure 2(b) shows details of the GCN Model. We use the python package PyG[2] to easily build a graph convolution network with the following propagation rule:

$$H^{(l+1)} = \sigma(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}) \tag{4}$$

Here, $\tilde{A} = A + I_N$ is the adjacency matrix of the undirected graph G with added self-connections. I_N is the identity matrix, $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$ and W^l is a layer-specific trainable weight matrix. $\sigma(.)$ denotes an activation function. $H^{(l)} \in R^{N \times D}$ is the matrix of activations in the l^{th} layer; $H^{(0)} = X$. Our forward model then takes the form below:

$$Z = \tilde{A}\ ReLU(\tilde{A}XW^{(0)})W^{(1)} \tag{5}$$

$$y = Sigmoid(W_s Z + b_s) \tag{6}$$

where W_s, b_s are parameters for Sigmoid layer,$W^{(0)} \in R^C$ is an input-to-hidden weight matrix for a hidden layer with H feature maps. $W^{(1)} \in R^R$ is a hidden-to-output weight matrix.

3) Fusion of Model: Assume the outputs of the model are o_1 and o_2 and the hyper parameter k , then the final probability distribution is computed as follows:

$$output = k \bigotimes o_1 + (1-k) \bigotimes o_2 \tag{7}$$

For the both models, we all use binary cross-entropy loss to optimize.

$$Loss(x_i, y_i) = -w_i[x_i log y_i + (1-x_i) log(1-y_i)] \tag{8}$$

where w_i is the parameter for loss, x_i is the i^{th} prediction of the label and y_i is the i^{th} ground truth.

IV. EXPERIMENTAL SETTINGS

A. Projects and dataset

We first use the CodeSplit[3] to split 100 high-quality Java projects on Github covering a variety of functions into method-level code fragments. Then we use Designite [18] to find out the smells contained in the source code and generate smell reports. Finally we choose nine code smells [18] at the method level for our experiment and combine their labels into a multi-label dataset. We divide all samples into three parts, 70% as the training set, 20% as the validation set, and 10% as the test set. Table I shows the number of samples used in our experiment and baselines.

B. Baseline

As mentioned before, we are the first to apply the deep learning methods to the multi-label code smell detection, and we select the following two improved methods as our baseline here, which are adapted into multi-label classification task:

1) Random–Forest Model: The model is used by [9] to reveal the multi-label classification and performs well when detecting *Long Method* and *Feature Envy*.

2) ASTNN Model: The ASTNN model is first introduced by Zhang [19] and was adapted by [8] to the single code smell detection. We refactor the model to do the multi-label code smell detection here.

[2]https://github.com/pyg-team

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

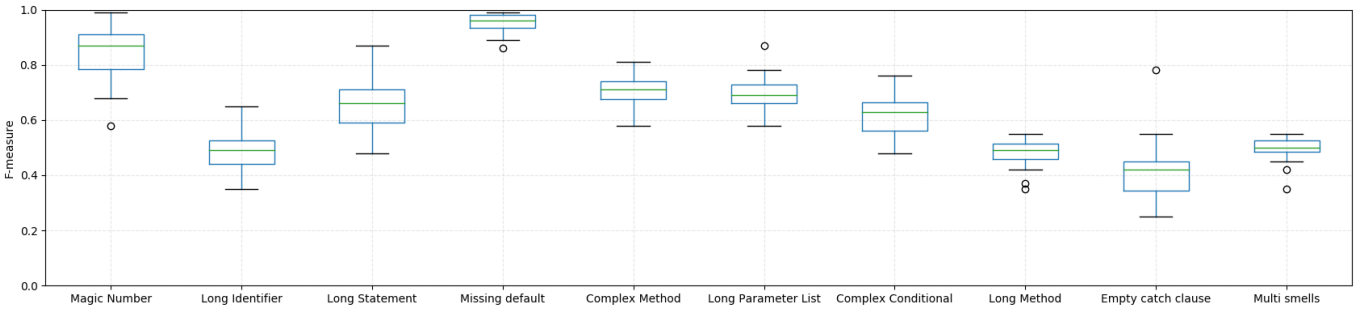

Fig. 3. Box plot of F-measure exhibit by HMML

TABLE I
SAMPLES DISTRIBUTIONS

	Training set		Validating set		Testing set	
Code smells	P	N	P	N	P	N
Magic Number	9643	76807	2748	21952	1387	10957
Long Identifier	926	85524	294	24406	132	12212
Long Statement	6816	69634	1955	22745	998	11346
Missing default	993	85457	308	24392	160	12184
Complex Method	2383	84067	713	23987	312	12032
Long Parameter List	1629	84821	33	24267	207	12137
Complex Conditional	1320	85130	346	24354	185	12159
Long Method	342	86108	92	24608	45	12299
Empty catch clause	558	85892	179	24521	75	12269
Multi smells	4972	81487	1475	23225	714	11630

C. 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. For multi-label code smell detection, we use the Macro weighted F1 [20], which considers the imbalance in the category of samples. $Precision_{weighted}$ and $Recall_{weighted}$ are weighted according to the number of categories. Assuming L is the number of categories, they are defined as follows:

$$Precision_i = \frac{True\ Positive_i}{True\ Positive_i + False\ Positive_i} \tag{9}$$

$$Precision_{weighted} = \frac{\sum_{i=1}^{L} Precision_i \times w_i}{|L|} \tag{10}$$

$$Recall_i = \frac{True\ Positive_i}{True\ Positive_i + False\ Negative_i} \tag{11}$$

$$Recall_{weighted} = \frac{\sum_{i=1}^{L} Recall_i \times w_i}{|L|} \tag{12}$$

$$F1_i = \frac{2 * Precision_i * Recall_i}{Precision_i + Recall_i} \tag{13}$$

$$Macro\ weighted\ F1 = \frac{2 * Precision_w * Recall_w}{Precision_w + Recall_w} \tag{14}$$

D. Training details

In our HMML method, the hidden states of LSTM have 300 dimensions and layer is set to be 2. We apply the graph convolution three times. In the two sub-models, the training batch size is set to be 32 and dropout is applied to avoid overfitting with dropout rate being 0.4. We use the Adam optimizer algorithm with 0.001 initial learning rate. In the ASTNN, we set two layers and 250 dimensions in the hidden states [19] Then we choose 80 features and 50 trees in the random forest [9]. Finally, we make our code public[4].

TABLE II
VALUES OF HYPER-PARAMETERS FOR HMML

Hyper-parameter	Values
Training batch size	{16,32,64,128}
Embedding dimensions(E)	{100,200,300}
Dimensions of hidden states in LSTM(H)	{150,250,300}
Number of layer in LSTM	{1,2,3}
Number of graph convolution in GCN	{1,2,3}

V. EXPERIMENTAL RESULTS

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

RQ1: How does our HMML method perform compared to other baselines?

RQ2: How does multi-label code smell detection perform compared with single code smell detection?

RQ3: What impact does each of our main components have in our HMML method?

A. RQ1:How does our HMML method perform compared to other baselines?

Table III shows the performance of our model and baselines on multi-label code smell detection and Figure 3 shows the box plot of performance of our HMML method under different configurations in Table II. From the table and figure, we can easily see that our model does not perform equally on all of the smells and it performs quite well on the smell like

[4]https://github.com/liyichen1234/HMML

TABLE III
PERFORMANCE OF HMML AND BASELINES ON MULTI-LABEL CODE SMELL DETECTION

	HMML			Random-Forest			ASTNN		
Code smells	P	R	F1	P	R	F1	P	R	F1
Magic Number	0.97	0.93	0.95	0.89	0.35	0.50	0.67	0.57	0.62
Long Identifier	0.44	0.55	0.49	0.52	0.36	0.43	0.85	0.30	0.44
Long Statement	0.73	0.60	0.66	0.90	0.35	0.50	0.84	0.68	0.75
Missing default	0.98	0.99	0.99	0.96	0.29	0.44	0.71	0.23	0.34
Complex Method	0.82	0.66	0.73	0.92	0.15	0.26	0.71	0.23	0.34
Long Parameter List	0.81	0.60	0.69	1.00	0.29	0.46	0.86	0.60	0.71
Complex Conditional	0.68	0.58	0.63	0.96	0.14	0.24	0.94	0.21	0.34
Long Method	0.67	0.41	0.51	1.00	0.09	0.16	0.83	0.69	0.76
Empty catch clause	0.51	0.30	0.38	0.86	0.08	0.15	0.61	0.11	0.18
Multi smells	0.83	0.74	0.78	0.90	0.30	0.45	0.76	0.52	0.62

Magic Number and *Missing default* and performs not bad on the other smells separately. In the meanwhile, the machine-learning method Random-forest has the poor Recall-values on each smell, which means the poor ability to find out the real code smell and the ASTNN performs unequally on the different smells.

In order to analyze the results, we apply the Win/Tie/Loss indicator to compare the performance of different models further, which has been used in prior works for performance comparison between different methods [8]. Then we conduct Wilcoxon signed-rank test and Cliff's delta test to analyze the performance of our model and other methods. Table IV shows Cliff's delta values($|\delta|$) and the corresponding effective levels.

To be specific, we make the following comparisons to determine the result of Win/Tie/Loss indicator: For a baseline method *M*, if our model outperforms *M* 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 models is statistical significant and can not be ignored. At this time, we mark our model as a "Win." In contrast, if the model *M* outperforms out model with a *p*-value $<$0.05 and a Cliff's delta$\geq$0.147, our model will be marked as a "Loss." Otherwise, we mark the case as a "Tie".

TABLE IV
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
$0.474 < \|\delta\|$	Large

As shown in the Table V, our method performs better almost in each smells and has an absolute advantage in multi code smells detection. Although weighted F1 can not accurately represent that the model can find all code smells at the same time, it reflects the ability of the model in multi-label code smell detection. Our HMML method has the highest weighted F1 and performs equally on the precision value and recall value. Therefore, we can regard that our HMML method does a good job in the multi-label code smell detection.

TABLE V
WIN/TIE/LOSS INDICATORS ON FMEASURE VALUES OF RANDOM FOREST, ASTNN, AND HMML

Code smell	Random Forest vs HMML	ASTNN vs HMML
Magic Number	$<$0.05(+Large)	$<$0.05(+Large)
Long Identifier	$<$0.05(+Medium)	$<$0.05(+Medium)
Long Statement	$<$0.05(+Large)	0.264(-Small)
Missing default	$<$0.05(+Large)	$<$0.05(+Large)
Complex Method	$<$0.05(+Large)	$<$0.05(+Large)
Long Parameter List	$<$0.05(+Large)	0.06(-Small)
Complex Conditional	$<$0.05(+Large)	$<$0.05(+Large)
Long Method	$<$0.05(+Large)	0.735(-Large)
Empty catch clause	$<$0.05(+Large)	$<$0.05(+Large)
Multi smells	$<$0.05(+Large)	$<$0.05(+Large)
Win/Tie/Loss	10/0/0	7/3/0

B. RQ2: How does multi-label code smell detection perform compared with single code smell detection?

TABLE VI
PERFORMANCE OF HMML AND BASELINES ON SINGLE CODE SMELL DETECTION

	HMML			Random-Forest			ASTNN		
Code smells	P	R	F1	P	R	F1	P	R	F1
Magic Number	0.98	0.94	0.96	0.88	0.36	0.51	0.85	0.85	0.85
Long Identifier	0.65	0.25	0.36	0.54	0.39	0.46	0.64	9.71	0.68
Long Statement	0.79	0.64	0.71	0.89	0.36	0.51	0.92	0.85	0.88
Missing default	0.88	0.84	0.86	0.93	0.34	0.50	0.83	0.72	0.77
Complex Method	0.84	0.83	0.84	0.84	0.15	0.26	0.85	0.25	0.39
Long Parameter List	0.86	0.72	0.79	1.00	0.50	0.66	0.86	0.58	0.70
Complex Conditional	0.79	0.50	0.61	0.93	0.14	0.24	0.83	0.75	0.79
Long Method	0.65	0.24	0.35	0.80	0.09	0.16	0.86	0.85	0.85
Empty catch clause	0.89	0.63	0.73	0.86	0.08	0.15	0.22	0.09	0.13

As shown in the Table VI, models show different abilities of detecting code smells. For each code smell, we apply the model used in multi-label code smell detection to train seperately and compare it with multi-label code smell detection model when detecting the designated code smell. Random-Forest performs equally in single code smell detection and multi-label code smell detection while ASTNN performs much better in single code smell detection. We believe this is somewhat related to capacity of ASTNN model, which cannot capture features of different code smells in the multi-label code smell detection. HMML performs slightly worse in multi-label code smell detection but still achieves a robust result.

What is more, single code smell detection needs to train corresponding model for each smell, which can be quite time consuming with the increase in the number of code smell. However, our multi-label code smell detection not only performs well in each code smell detection but can find out all code smells by one model at the same time.

C. RQ3: What impact does each of our main components have in our model?

We analyze the performance gain achieved due to various components of our approach by performing an ablation study.

Table VII shows these results. Control and data flow edges play a major role in the code smell detection. The reason is that code smells like *Empty catch clause* and *Missing default* which have the complex data flow information can be found out effectively in the GCN model. We can also find that the LSTM model and GCN model all perform bad on *Long Method*. This is because *Long Method* needs not only structural information but syntactic and semantic information.

TABLE VII
EFFECTIVENESS OF EACH MODULE IN HMML

	HMML^{-GCN}			HMML^{-LSTM}			HMML$^{-control\ and\ data\ flow\ edges}$		
Code smells	P	R	F1	P	R	F1	P	R	F1
Magic Number	0.98	0.89	0.93	0.87	0.85	0.86	0.83	0.81	0.82
Long Identifier	0.62	0.14	0.22	0.12	0.75	0.21	0.11	0.76	0.19
Long Statement	0.75	0.51	0.61	0.61	0.79	0.69	0.59	0.72	0.65
Missing default	0.89	0.79	0.84	0.99	0.96	0.97	0.77	0.74	0.73
Complex Method	0.90	0.59	0.71	0.75	0.66	0.70	0.70	0.63	0.66
Long Parameter List	0.80	0.41	0.54	0.85	0.71	0.77	0.86	0.73	0.79
Complex Conditional	0.77	0.58	0.66	0.77	0.58	0.66	0.75	0.42	0.54
Long Method	0.56	0.11	0.19	0.57	0.18	0.27	0.48	0.22	0.30
Empty catch clause	0.50	0.04	0.08	0.85	0.70	0.77	0.74	0.62	0.67
Multi smells	0.86	0.65	0.74	0.75	0.78	0.75	0.71	0.72	0.71

Fortunately, HMML notices the advantage and disadvantage of each model, which means the ability to catch appropriate features in multi-label code smell detection. HMML balances the results with the fusion of models and achieves a more robust result.

VI. THREATS TO VALIDITY

A. Internal validity

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. The tool uses three quotes to get more than 20 labels. Although the tool has been applied to many related works, it still needs much time to ensure the reliability of data.

B. External validity

We just did our detection on the 100 Java projects on Github. More jobs should be carried out on other projects, even transfer the model to the other languages since different languages may have its own distribution of code smells.

VII. CONCLUSION AND FEATURE WORK

In this paper, we propose a hybrid model, which extracts the multi-level code representation information to reveal multi-label code smell detection. Then we carry out the experiment based on the deep learning method and achieve a good result in terms of the evaluation.

As future work, a unified framework to deal with code smells at different granularities should be considered and we want to figure out whether existed approaches have the ability to find the unknown code smell. Moreover, it is of great value to make the model feasible to other programming languages.

VIII. ACKNOWLEDGEMENT

This work is partially supported by the National Natural Science Foundation of China (61772263, 61872177), Collaborative Innovation Center of Novel Software Technology and Industrialization, and the Priority Academic Program Development of Jiangsu Higher Education Institutions.

REFERENCES

[1] M. Fowler, *Refactoring - Improving the Design of Existing Code*, ser. Addison Wesley object technology series. Addison-Wesley, 1999. [Online]. Available: http://martinfowler.com/books/refactoring.html

[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] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1143–1191, 2016.

[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 (saner)*. IEEE, 2018, pp. 612–621.

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

[8] W. Xu and X. Zhang, "Multi-granularity code smell detection using deep learning method based on abstract syntax tree," in *Proceedings of the 33rd International Conference on Software Engineering and Knowledge Engineering (SEKE)*, 07 2021, pp. 503–509.

[9] T. Guggulothu and S. A. Moiz, "Code smell detection using multi-label classification approach," *Software Quality Journal*, vol. 28, no. 3, pp. 1063–1086, 2020.

[10] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting code clones with graph neural network and flow-augmented abstract syntax tree," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 261–271.

[11] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

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

[13] 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.

[14] T. Sharma, V. Efstathiou, P. Louridas, and D. Spinellis, "Code smell detection by deep direct-learning and transfer-learning," *Journal of Systems and Software*, vol. 176, p. 110936, 2021.

[15] 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.

[16] 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.

[17] X. Chen, C. Liu, and D. Song, "Tree-to-tree neural networks for program translation," *arXiv preprint arXiv:1802.03691*, 2018.

[18] 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.

[19] 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.

[20] M. Grandini, E. Bagli, and G. Visani, "Metrics for multi-class classification: an overview," *arXiv preprint arXiv:2008.05756*, 2020.