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: xfang@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 hybrid model with multi-level 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
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 separately 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
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.

\[ c_j = \sum_{i=1}^{x} a_{ij} h_i \]  

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.

\[ r_{ij} = h_i \ast c_j \]  
\[ y = \text{Sigmoid}(W_s r_{ij} + b_s) \]

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:

\[ H^{(l+1)} = \sigma(\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} H^{(l)} W^{(l)}) \]

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. \( \sigma(\cdot) \) 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:

\[ Z = \tilde{A} \text{ ReLU}(\tilde{A} X W^{(0)}) W^{(1)} \]
\[ y = \text{Sigmoid}(W_s Z + b_s) \]

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:

\[ output = k \otimes o_1 + (1 - k) \otimes o_2 \]  

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

\[ Loss(x_i, y_i) = -w_i [x_i \log y_i + (1 - x_i) \log (1 - y_i)] \]

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
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<sub>weighted</sub> and Recall<sub>weighted</sub> are weighted according to the number of categories. Assuming L is the number of categories, they are defined as follows:

\[
\text{Precision}_i = \frac{\text{True Positive}_i}{\text{True Positive}_i + \text{False Positive}_i}
\]

\[
\text{Precision}_\text{weighted} = \frac{\sum_{i=1}^{L} \text{Precision}_i \times w_i}{|L|}
\]

\[
\text{Recall}_i = \frac{\text{True Positive}_i}{\text{True Positive}_i + \text{False Negative}_i}
\]

\[
\text{Recall}_\text{weighted} = \frac{\sum_{i=1}^{L} \text{Recall}_i \times w_i}{|L|}
\]

\[
F_1 = \frac{2 \times \text{Precision}_i \times \text{Recall}_i}{\text{Precision}_i + \text{Recall}_i}
\]

\[
\text{Macro weighted F1} = \frac{2 \times \text{Precision}_\text{weighted} \times \text{Recall}_\text{weighted}}{\text{Precision}_\text{weighted} + \text{Recall}_\text{weighted}}
\]

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<sup>4</sup>.

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

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

<sup>4</sup>https://github.com/liyichen1234/HMML
Random Forest vs HMML

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

As shown in 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 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>
<thead>
<tr>
<th>Code smells</th>
<th>HMML + GCN</th>
<th>HMML + LSTM</th>
<th>HMML + control and data flow edge</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P</td>
<td>R</td>
<td>F1</td>
</tr>
<tr>
<td>Magic number</td>
<td>0.98</td>
<td>0.89</td>
<td>0.93</td>
</tr>
<tr>
<td>Long identifier</td>
<td>0.62</td>
<td>0.14</td>
<td>0.22</td>
</tr>
<tr>
<td>Long statement</td>
<td>0.75</td>
<td>0.51</td>
<td>0.61</td>
</tr>
<tr>
<td>Missing default</td>
<td>0.89</td>
<td>0.79</td>
<td>0.84</td>
</tr>
<tr>
<td>Complex method</td>
<td>0.90</td>
<td>0.59</td>
<td>0.71</td>
</tr>
<tr>
<td>Long parameter list</td>
<td>0.80</td>
<td>0.41</td>
<td>0.54</td>
</tr>
<tr>
<td>Complex conditional</td>
<td>0.77</td>
<td>0.58</td>
<td>0.66</td>
</tr>
<tr>
<td>Long method</td>
<td>0.56</td>
<td>0.11</td>
<td>0.19</td>
</tr>
<tr>
<td>Empty catch clause</td>
<td>0.50</td>
<td>0.04</td>
<td>0.08</td>
</tr>
<tr>
<td>Multi smells</td>
<td>0.86</td>
<td>0.65</td>
<td>0.74</td>
</tr>
</tbody>
</table>

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