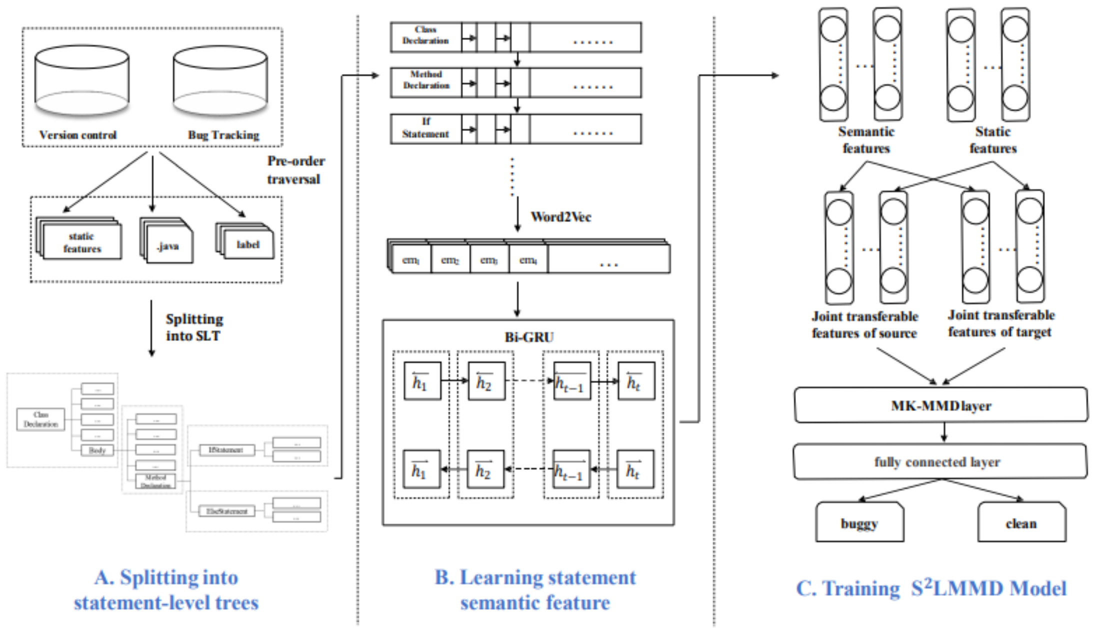
S^2LMMD: Cross-Project Software Defect Prediction via Statement Semantic Learning and Maximum Mean Discrepancy

Wangshu Liu[†‡*] , Yongteng Zhu[†], Xiang Chen[§‡], Qing Gu[‡], Xingya Wang[†‡], Shenkai Gu[†]
[†]School of Computer Science and Technology, Nanjing Tech University, Nanjing, China
[‡]State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
[§]School of Information Science and Technology, Nantong University, Nantong, China



A. Splitting into statement-level trees

For each Java source file in a given project, we extract their abstract syntax trees by a third-party python library javalang. As suggested by Zhang et al, a statement tree, which represents a code statement instead of an entire code file, outperforms a full AST when learning the code semantic and structural features. Therefore, we first construct the statement level trees (SLT). The entry definition of the statements is essential for converting a parsed AST into several SLTs. In our study, we divide the statement entry into four categories:

•The root of the original AST

•The declaration node (e.g., PackageDeclaration, Class Declaration, and MethodDeclaration)

• The control-flow node (e.g., IfStatement, ForStatement, and TryStatement)

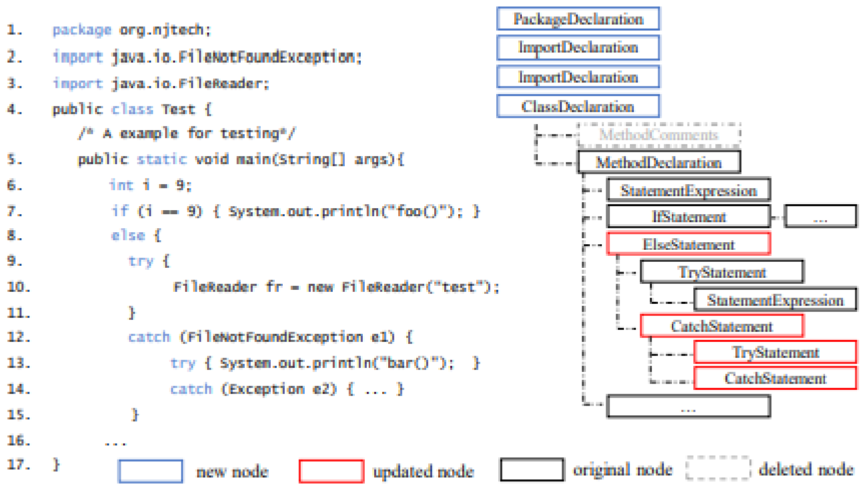
• The executable statement node (e.g., StatementExpression)

The AST always starts with a statement entry, and can be defined as a collection of multiple entries $\{e_1, e_2, \cdots, e_n\}$, where e_i represents the i^{th} entry in AST. Then an AST can be split into n SLTs, where SLT_i is generated by all the nodes from the entry e_i to the next node of e_{i+1}. Noted that the last SLT is formed from e_n to the end node.

Compared to the previous study, we make the following improvements: (1) We apply

SLT splitting technology into the class granularity, which is more practical under a real-world project, rather than the method granularity. (2) The previous study ignored the nesting problem between control-flow statements. For example, by their method, the “if-else” statement will be transformed into two “if-block”s, but the correct alternative is “if-block” followed by “else-block”. The same problem can be found in the “catch-try” nesting statement. It should be transformed into “catch-block” followed by “try-block” instead of embedding “try” in the previous “catch-block” statement. (3) We filter all comments, including the class comment and the method comment, which are not helpful to improve the performance of the defect prediction. Meanwhile, we treat each import statement as a single SLT, since the incorrect library import may lead to software defects.

The example of our proposed SLT splitting method is shown in Fig. From Lines 1-4 of the program snippet, different with the previous study, we can handle ImportDeclaration and ClassDeclaration, which are common as the standard element in the source code. Moreover, for Line 8-9, our method can detect “else” node and simultaneously build two SLTs to represent (i.e., ElseStatement and TryStatement). Similarly, the “catch-try” sequence in Lines 12-13 can be dealt with as TryStatement after CatchStatement, while the “try” in “catch-try” is not considered as an independent statement in Zhang’s method. Finally, in the snippet of Fig, our method can generate 14 SLTs at least.


```
1.  package org.njtech;
2.  import java.io.FileNotFoundException;
3.  import java.io.FileReader;
4.  public class Test {
      /* A example for testing*/
5.    public static void main(String[] args){
6.        int i = 9;
7.        if (i == 9) { System.out.println("foo()"); }
8.        else {
9.          try {
10.               FileReader fr = new FileReader("test");
11.         }
12.         catch (FileNotFoundException e1) {
13.               try { System.out.println("bar()");  }
14.               catch (Exception e2) { ... }
15.          }
16.       ...
17. }
```

B. Learning statement semantic feature

Because the tree structure of SLT cannot be direct as the input for neural networks, we should preprocess the SLT data in the first embedding layer. We adopt a pre-order traversal method to convert all tree structures into sequence vectors. The maximum sequence length is fixed as the size of the vector, and if the sequence length is insufficient, zero values are filled to the sequence vector. Thereafter, we encode and optimize the sequence vectors by an open-source PyTorch toolkit gensim.

After the embedding layer, we construct the Bi-GRU layer. Gated recurrent unit (GRU) is a kind of recurrent neural network (RNN) and has been widely used to capture the semantic context in natural language processing. Compared with LSTM, GRU has fewer gate layers and fewer parameters that need to be optimized. In this study, we employ the Bi-GRU model, which is composed of two parallel GRU layers, to learn the semantic and structural information of both the past SLT and the future SLT from the back and forth direction.

Generally, a GRU layer contains one update gate Z for updating the hidden states and one reset gate R for deciding which states should be forgotten. Assuming that a source code can be represented as a matrix $EM = [\mathrm{em}_1, \mathrm{em}_2, \cdots, \mathrm{em}_n] \in \mathbb{R}^{n \times d}$ after the embedding layer, where d represents the word embedding dimension. For a given time t, the current embedding em_t and the previous hidden state H_{t-1} are the inputs of both update gate Z and reset gate R. The output H_t of the GRU layer can be calculated by the following formula:

$$Z = \sigma(\mathrm{em}_t W_{em}^{(Z)} + H_{t-1} W_H^{(Z)} + b^{(Z)})$$

$$R = \sigma(\mathrm{em}_t W_{em}^{(R)} + H_{t-1} W_H^{(R)} + b^{(R)})$$

$$\widetilde{H_t} = \tanh(\mathrm{em}_t W_{em} + (R \odot H_{t-1}) W_H + b)$$

$$H_t = (1 - Z) \odot H_{t-1} + Z \odot \widetilde{H_t}$$

Where $\sigma(\cdot)$ and $tanh(\cdot)$ are activation functions, $\odot$ stands for element-wise product. $W_{em}^{(Z)}$, $W_{em}^{(R)}$, $W_{em} \in \mathbb{R}^{d \times h}$, and $W_H^{(Z)}$, $W_H^{(R)}$, $W_H \in \mathbb{R}^{h \times h}$ are weight parameters that need to be learned. $b^{(Z)}$, $b^{(R)}$ and b are bias units. h is the dimension of the hidden layer.

$$\overrightarrow{H_t} = \overrightarrow{GRU}(\mathrm{em}_t, \overrightarrow{H_{t-1}})$$

$$\overleftarrow{H_t} = \overleftarrow{GRU}(\mathrm{em}_t, \overleftarrow{H_{t-1}})$$

$$H_t = [\overrightarrow{H_t}, \overleftarrow{H_t}]$$

Furthermore, as shown in Formula, to build Bi-GRU for learning semantic features, we input the original order of EM for one GRU layer and fetch a reverse order for another GRU. After employing the max-pooling operation, we finally obtain a statement semantic vector within $\mathbb{R}^{2h}$, which is the learned semantic feature for building a CPDP model in the next phase.

C. Training S^2LMMD model

For building a robust model and enriching our features, we joined the static program features f_{static} in PROMISE repository, which has been widely used in previous studies. We also use the min-max normalization method on the static features, since the different scale unit may mask the effect of certain features. After appending static program features, the dimension of all features becomes to $2h + |f_{static}|$.

Let $D_s = \{x_i^s, y_i^s\}_{i=1}^m$ and $D_t = \{x_j^t, y_j^t\}_{j=1}^n$ represent the source project and the target project respectively. To construct a CPDP model, we first calculate the classification error by the automatically generated semantic features joint with the static program features. Here, we prefer to use the typical cross entropy as the loss function, and formally write the loss function $\mathcal{L}_{S^2L}$ as Formula.

$$\mathcal{L}_{S^2L} = \frac{1}{m} \sum_{i=1}^{m} -[y_s^i \cdot \log\big(\ominus(x_i^s)\big) + (1 - y_i^s) \cdot \log(1 - \ominus(x_i^s))]$$

where $\ominus$ denotes all of the weights and bias learned in our model.

Moreover, to alleviate the distribution difference between the source project and the target project, we employ a transfer loss function with maximum mean discrepancy (MMD). MMD uses a kernel-trick method to map the original features into the high-dimensional reproducing kernel Hilbert space (RKHS) and can be calculated as follows.

$$\begin{aligned}\mathcal{L}_{MMD} &= MMD(D_s, Dt) \\ &= \left\| \frac{1}{m}\sum_{i=1}^{m} \emptyset(x_i^s) - \frac{1}{m}\sum_{j=m+1}^{m+n} \emptyset(x_j^s) \right\|_H^2 \\ &= \left\| \frac{1}{m^2}\sum_{i=1}^{m}\sum_{j=1}^{m} \mathcal{K}\left(x_i^s, x_j^s\right) + \frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n} \mathcal{K}\left(x_i^t, x_j^t\right) - \frac{2}{mn}\sum_{i=1}^{m}\sum_{j=1}^{n} \mathcal{K}\left(x_i^s, x_j^t\right) \right\|_H\end{aligned}$$

where $\|\cdot\|_H$ stands for the RKHS norm, and $\emptyset(\cdot)$ is the feature mapping, which is related to the kernel mapping $\mathcal{K}\left(x_i^s, x_j^t\right) =< \emptyset(x^s), \emptyset(x^t) >$. In this study, we use Gaussian kernel $e^{-\frac{\|x-y\|^2}{2\sigma^2}}$ as our kernel function, and utilize nine different σ to combine a multiple kernel MMD for describing the distribution divergence more flexible.

Therefore, the final loss function in our proposed method $\mathrm{S^2LMMD}$ can be estimated by Formula

$$\mathcal{L} = \mathcal{L}_{S^2L} + \lambda\mathcal{L}_{MMD}$$

We train the model under the mini-batch stochastic gradient descent (SGD) algorithm using the Adam optimizer. By putting the new program modules into our trained model, the probability will be used for determining which module may contain defects.