A Deep Method Renaming Prediction and Refinement Approach for Java Projects

Jiahui Liang[1], Weiqin Zou[1], Jingxuan Zhang[1], Zhiqiu Huang[1], Chenxing Sun[2]
[1]*College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China*
[2]*Tencent Technology Company Limited, Shenzhen, China*
{liangjiahui, weiqin, jxzhang, zqhuang}@nuaa.edu.cn marssun@tencent.com

***Abstract*—During the process of software development and maintenance, developers would regularly refactor existing source code to improve efficiency and maintainability. Among various code refactoring activities, method renaming often happens within the whole project evolution process. To perform method renaming, developers should first identify the exact methods that should be renamed, which is generally tedious and error-prone through manual analysis. Towards this end, researchers have proposed some approaches to automatically recommend candidate methods for renaming. To further improve the performance of existing techniques, in this paper, we propose a novel approach that fully leverages historical code changes and overlapping relationships among code entities to identify renaming opportunities for methods. Specifically, we first embed methods into vectors and incorporate overlapping relationships among code entities by using different attention heads in a deep learning network. Then, we apply these obtained vectors to train a classifier to predict potential renaming opportunities for methods. Finally, we utilize historical renaming activities of related code entities to further refine the predicted results. Experimental results on 114,398 methods from 10 open source Java projects show that our approach could outperform the state-of-the-art approach by achieving an average F-measure of 80.02%. To better validate the effectiveness of our approach, we also explore the performance of some major components of our approach. For example, we find that employing related code entities help to improve the performance of our approach by 40.40% in terms of the average F-measure.**

***Index Terms*—Software Refactoring, Renaming Opportunities, Code Change History, Representation Learning**

I. Introduction

Code refactoring is the process of restructuring existing internal source code without changing the external software behaviors. The aim of code refactoring is to improve the non-functional attributes of source code, e.g., design and implementation, while preserving the main function of the software. In the practical software development process, developers frequently perform code refactoring activities [1], [2] to improve code readability and reduce code complexity [2]–[6].

Among the various types of code refactoring, code-entity (e.g., methods, variables) renaming is one of the most important and common code refactoring activities [7], [8], which mainly aims to make the literal meanings of code entities consistent with the corresponding semantic functions [2], [5], [9]. Among various code-entity renamings, method renaming is of particular importance given that methods are generally taken as basic function units of source code but are frequently reported to show inconsistencies with method implementations by existing studies [5], [10]–[13]. One of the key issues for method renaming is to identify method renaming opportunities, i.e., identifying the exact methods which need to be renamed. Manually identifying method renaming opportunities is time-consuming and error-prone. For example, method renaming often occurs in the maintenance phase, especially in the version upgrade. A developer may spend a lot of time to understand the program if he/she tries to do such method renamings for the first time. Moreover, in large projects with complex code-entity relationships, method renaming may be interlocking and missing any renaming would make the well-running program fail to behave correctly. Therefore, it is necessary and worthwhile to detect method renaming opportunity.

To help developers better perform method renaming refactorings, some researchers have proposed several approaches to automatically identify potential method renaming opportunities [5], [10]–[12], among which the technique proposed by Liu et al. [10] is the state-of-the-art one. The approach proposed by Liu et al. [10] identify renaming opportunities by leveraging conducted renaming activities. Once a renaming activity is conducted manually or with tool support, the proposed approach recommends to rename closely related code entities whose names are similar to that of the renamed entity. However, there is still much room for improvement in the following two aspects.

First, we empirically find that refactoring activities are related to each other. However, the approach proposed by Liu et al. [10] only relies on a single initial refactoring activity and propagates it to the other related code entities without considering the other refactoring activities. The performance of renaming opportunity identification can be further improved if the historical refactoring activities of all the related code entities can be fully leveraged. Second, the approach proposed by Liu et al. [10] lacks deep semantic understanding to the source code, especially the semantic overlapping relationship between code entities. The overlapping relationships among code entities mean that one code entity may participate in multiple relationships in the same method implementation [14]. For example, Fig. 1 shows a real-world example, which is a method that returns the maximum value. For this method, the method name *getMaxInteger* has a call relationship with the code entity *println* method. For another example, it

Jingxuan Zhang is the corresponding author.

```
public static int getMaxInteger(int num1, int num2) {
  int result;
  if (num1 > num2){
    result = num1;
  }else{
    result = num2;
  }
    System.out.println(result);
    return result;
}
```

Fig. 1. An example to show overlapping relationships among code entities.

also has a constraint relationship with the code entity *int*. During the semantic analysis of the code, the overlapping relationships can be beneficial for the detection of method renaming opportunities. Hence, the overlapping relationships among code entities should be better utilized for improving the semantic understanding of source code and further improving the performance of identifying renaming opportunities.

In order to better identify method renaming opportunities, we propose a novel approach using the pre-trained model Bidirectional Encoder Representations from Transformers (BERT) and text Convolutional Neural Network (textCNN), equipped with a prediction result adjusting component through historical renaming activities analysis [15], [16]. Specifically, considering that BERT has its natural advantage in capturing the deep context semantics of textual contents [8], [15], [17]–[20], we decide to use BERT to capture the lexical and overlapping relationships among code entities. More detailedly, we first employ BERT to embed method implementations into numeric embedding vectors, and then use textCNN to further enhance these embedding representations. Based on these enhanced representations, we train a classifier to predict potential renaming opportunities. After obtaining the initial renaming opportunities, we leverage the historical renaming activities of related code entities to further refine the prediction results.

Experimental results on 114,398 methods from 10 open source Java projects show that our approach achieves an average F-measure of 80.02%, which outperforms the state-of-the-art approach by 4.94% on average. Further, we find that historical renaming activities could greatly refine the prediction results by improving the whole F-measure by 40.40%. We also investigate the influence of the size of the training corpus on the performance of our approach, we find that along with the increase of the size of training corpus, our approach achieves better results generally. In addition, we also find that even relying on a small training corpus, our approach could also achieve comparable good results.

Our major contributions are as follows.

- We propose a novel approach that fully leverages historical renaming activities and overlapping relationships among code entities to help identify method renaming opportunities.
- We conduct extensive experiments to show the effectiveness of our approach. Experimental results demonstrate that our approach outperforms the state-of-the-art approach.
- We open source our technique and share the experimental datasets to the public[1] for reproduction and inspiring further research in method renaming identification.

The remaining part of this paper is structured as follows. In Section II, we describe the background knowledge of some techniques used in this study. Next, we illustrate our approach in Section III. Then, we detail the experimental setup and experimental results in Sections IV and V respectively. We discuss the threats to validity and related work in Sections VI and VII. Finally, we summarize our work in Section VIII.

II. Background

In this section, we provide a brief introduction about the word embedding technique and the BERT language model used within our approach as follows.

A. Wording Embedding

Word embedding is a technique that encodes words and sentences into dense vectors with a fixed length. The core idea of the word embedding technique is to train a Neural Network Language Model (NNLM) that predicts the masked word based on its Context-Before and Context-After contents and take word vectors as its incidental outputs [21]–[23]. In this study, we mainly employ the word embedding technique within the state-of-the-art language model BERT [15] to transform code entities into vectors.

B. BERT Language Model

BERT is a language model that uses a bi-directional transformer encoder to generate deep bi-directional language representations [15]. The pre-processed data is used as the original input of BERT, and the input of the BERT encoding layer is the sum of the three embeddings learned from the three embedding layers. As the key part of BERT, transformer [24] includes an encoder and a decoder. The decoder is with an additional sub-layer of attention compared against LSTM [25]. Since BERT could well capture the context semantics of terms [26], we decide to use BERT to capture the semantic features (e.g., the overlapping relationships of code entities) of methods to help us identify method renaming opportunities in this paper.

III. Methodology

In this section, we mainly describe the framework of our approach for identifying method renaming opportunities. In a word, as shown in Fig. 2, we regard the identification of method renaming opportunities as a classification task, which includes three parts, namely data pre-processing, model training, and method renaming prediction&correction. More details are as follows.

[1]https://github.com/konL/MethodRenamePrediction

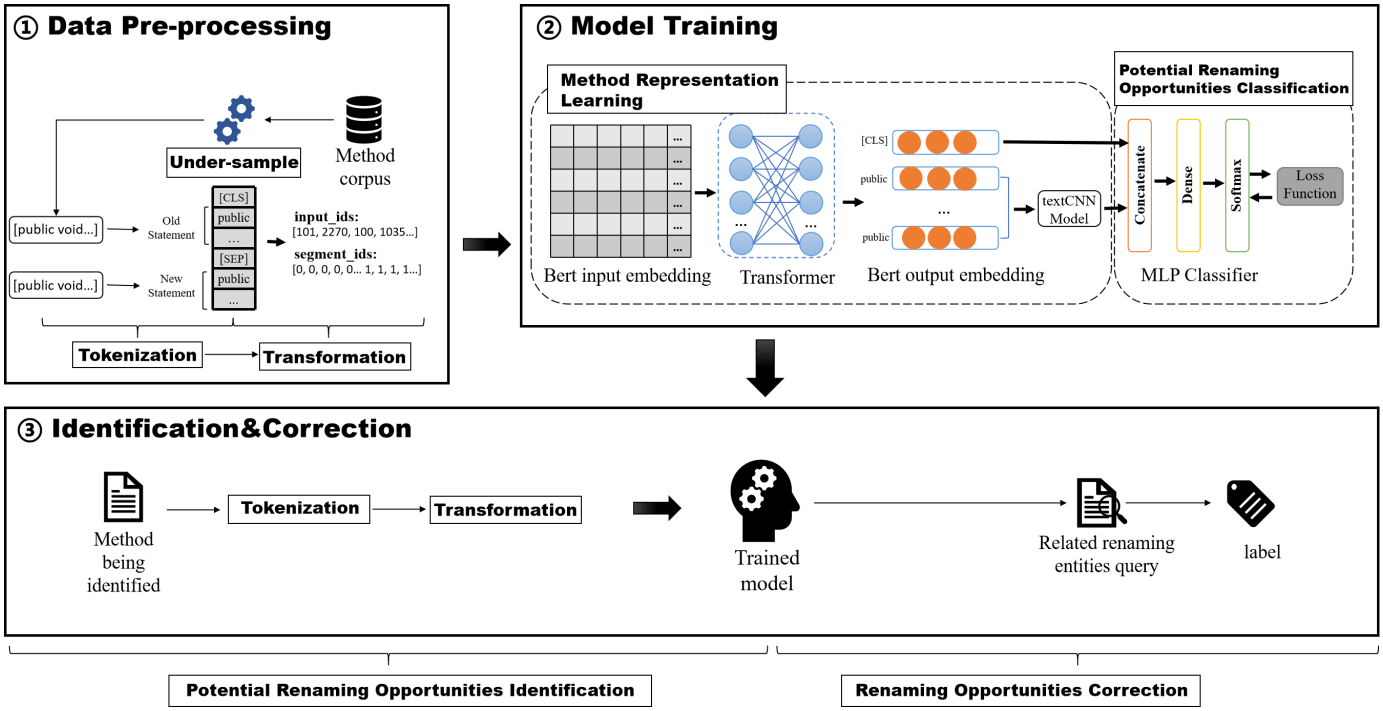

Fig. 2. The workflow of our approach to identify renaming opportunities.

A. Data Pre-processing

As mentioned in the Introduction section, we adopt BERT to capture the semantic features of methods. Before applying BERT, we need to first preprocess the raw method data to make them work as suitable inputs for BERT. Since the number of methods needed for renaming is generally much smaller than that with no renaming needed, to get a well-performed prediction model, we employ an under-sampling strategy to handle the imbalanced class problem. After obtaining the sampled methods, we perform tokenization and towards them to get the final input for BERT. The details of sampling and tokenization are as follows.

1) Under-Sampling: We adopt the random under-sampling strategy[2] to solve the imbalanced class problem. In random under-sampling process, instances of the majority class (i.e., methods without renaming) would be repeatedly filtered out till that the number of majority class is equal to that of the minority class (i.e., methods with renaming performed). By using the random under-sampling strategy, all information in the minority class could be reserved, meanwhile, the weights of the majority class could be relatively weakened. Since the minority class is much more of our interest than the majority class, we believe that the adoption of under-sampling is reasonable in our experimental settings.

2) Tokenization: For each sampled method, we retrieve its code as a string and use spaces and punctuation to split it into a list of tokens. Meanwhile, for those tokens in camel case, we further split them by the capital characters (e.g., the token 'getMax' is split into two tokens, i.e., 'get' and 'Max'). All tokens would be converted into lower case for further processing.

B. Model Training

The model training part mainly includes three components, namely training dataset construction, method semantic representation, and Multiple Layer Perception (MLP) classifier application.

1) Training Dataset Construction: Before training a method-renaming classifier, we need to first retrieve relevant methods from software projects to construct a dataset for model building. Specifically, we first collect the historical and current versions (i.e., code versions before and after method renaming) of all files with methods ever renamed. Then, we use JavaParser[3] to extract the method name and method body in the files from the historical and current versions respectively, with extracted two versions of methods being a method pair. For each method pair, we check whether the methods have ever been renamed by analyzing the historical and current versions of methods. In other words, if the old (historical) method name and new (current) method name are the same, it means that the method name has not been changed; while if two method names are not exactly the same, it indicates that the method name has been changed (renamed). In this way, we can determine the class label (i.e., renamed or not) for each method in the training dataset.

[2]https://imbalanced-learn.org/stable/under_sampling.html

[3]https://github.com/javaparser/javaparser

2) Method Semantic Representation: After collecting relevant methods, we apply the pre-trained BERT language model [15] and textCNN [16] to capture the semantics of methods. Detailedly, as mentioned in Section II, the BERT network would use three embedding layers, namely token embedding, segment embedding, and position embedding, to capture the semantics of methods from different aspects. After obtaining the three embeddings, BERT would use the following formula to merge them into a new embedding vector (in the formula, token, segment, and position represents token embedding vector, segment embedding vector, and position embedding vector respectively).

$$Z = D_{token} + D_{segment} + D_{position} \tag{1}$$

The embedding representation *Z* would be further passed to the encoder of the transformer component of BERT, and generate a new embedding vector through the decoder of the transformer. Further, we apply textCNN towards the embedding outputs of BERT to better capture the semantic features of methods. After applying textCNN, we then get the embedding vectors of methods based on which a classifier could be built.

3) MLP Classifier Application: After we obtain the semantic features of methods represented in numeric embedding vectors, we start to build a MLP model (a classic feedforward artificial neural network) based on the embedding vector dataset for method renaming prediction. A MLP model would generally contain an input layer, a hidden layer and an output layer. In our approach, the embedding vectors of methods are fed into the input layer. Then, we use a fully-connected hidden layer with 512 nodes to further retrieve high-level semantics of the inputs. Last, we use the logistic classifier as the output layer to predict the renaming label for a given method, with *Softmax* as the activation function.

C. Method Renaming Prediction & Correction

This part includes two components, i.e., potential renaming opportunities prediction and renaming opportunities correction. The potential renaming opportunities identification component aims to use the trained model to predict whether a given method (represented in a numeric embedding vector) needs to be renamed (i.e., the initial prediction results); while the renaming opportunities correction component aims to leverage historical renaming activities of related code entities to refine the initial prediction results.

1) Potential Renaming Opportunities Prediction: For a given new method *m*, we can obtain the word index sequence and segment sequence [27] of its historical and current implementations (i.e., the versions before and after renaming) after data pre-processing. Then, these sequences are fed into the BERT model to generate embeddings representing the semantic relationships among code entities by using the following formula:

$$V = BERT\_enhanced(m) \tag{2}$$

where $BERT\_enhanced(*)$ is the function to obtain the final embedding representation V for the method. We take the embedding V of the given method *m* as the input to the MLP classifier. The MLP classifier would output the class label for the method as follows:

$$prediction = classifier(V) \tag{3}$$

where $classifier(*)$ is a binary classifier. If the output is true, then it means the method need renaming. Otherwise, there is no need for renaming of this method.

2) Renaming Opportunities Correction: After we obtain a list of methods predicted as needing renaming, we attempt to leverage the historical renaming activities of code entities related to the methods for further prediction results refinement. The historical code entity relationships are used as intermediate information to filter out historical renaming activity. Such a design could avoid the analysis of complex code relationships and uses a simpler form (e.g., rules) to obtain the key information needed for classification. Here, for a given method, its related code entities include the class it belongs to, its caller methods in the same file, and those methods called within the method body. Specifically, if a method is predicted as needing renaming by the trained model, we would check whether its related code entities have ever been renamed before. If those code entities have never been renamed in code revision history, then we determine that there is no need to rename the target method. Otherwise, the target method would be output as renaming candidates for refactoring.

IV. Experimental Setup

In the experimental setup section, we mainly introduce the process of data collection, the baseline approach which our technique is compared against, the evaluation method, and the evaluation metrics in details.

A. Data Collection

We collect the experimental dataset from 10 open-source Java projects with the highest number of stars hosted in the Apache community. Those projects are popular among users/developers; they are from different domains, of different code-scales and have a sufficient number of commit messages. Such characteristics make us believe that the performance of our approach on these projects would project valuable insights into the application of our approach in real software development practice to a certain extent. Table I provides more details of our experimental projects. The main data collection procedures include identifying historical renamed methods from these 10 projects and collecting and filtering renamed methods (with implementations) for experiments.

1) Identifying Historical Renamed Methods: First, we employ JavaParser and the git log command *git log -L start,end:file* to obtain the change history of methods based on their start and end line number. An example of the git log command is shown in Fig. 3. The boxes in Fig. 3 give examples of the detailed code change information. Then, we iterate over all the commits chronologically. If the similarity

TABLE I
THE MAIN CHARACTERISTICS OF THE SELECTED PROJECTS RANKED BY STARS.

Project	Git SHA	Stars (K)	Files	All methods	Filtered methods	Ave. token length	Positive methods	Negative methods
dubbo	7dd685	35.7	680	568	122	72.04	20	102
flink	03ca39	16.6	11,730	5,140	786	66.23	102	684
cassandra	dbf6e6	6.7	2,993	9,963	744	72.81	228	516
storm	3f96c2	6.3	2,418	159,496	83,944	83.08	143	83,801
tomcat	5ae107	5.4	2,513	67,151	5,803	31.03	2,800	3,003
jmeter	3bd28d	5.4	1,386	5,413	744	72.61	258	486
zeppelin	985bb0	5.3	947	21,574	11,871	70.82	163	11,708
beam	b9bb2a	4.9	4,790	6,883	1,634	66.43	295	1,339
hbase	9d5004	4.1	4,361	96,958	6,336	73.33	722	5,614
camel	110071	3.8	19,991	19,050	2,414	66.77	633	1,781

Fig. 3. An example of the git log command.

between the current statement and the historical statement is greater than a threshold value (0.85 in this paper) and the method names contained in both statements are not the same, we determine that it is indeed a refactoring activity. In this study, we use Levenshtein distance to calculate the similarity, which is also used in Sheneamer's study [28] to detect the similarity between two blocks of code. The similarity threshold is set following the parameters setting proposed by Sheneamer et al. [28]. For each iteration, we obtain the historical information (e.g. historical statement and historical method name) in the commit message. Then, we regard the historical statement as the current statement and repeat the above steps until all the commits are checked. Finally, we could obtain those methods which have ever been renamed after we finish checking the historical changes for all methods.

2) Collecting and Filtering Renamed Methods: In the previous subsection, we could identify methods with renaming activities. For each renamed method, with the help of JavaParser, we further download two versions of source code files that contain the method, namely the files before and after renaming the method. All the method implementations from those code files are then collected as a method corpus for further filtering. In this step, we collect a total of 392,196 methods from ten experimental projects.

Given that our approach heavily relies on token-related information in method implementations, we filter those methods with empty method body out from our dataset. Meanwhile, we also remove those methods whose implementations have never been changed along with the project evolution. Finally, 114,398 methods from ten projects are left for experiments.

B. Baseline

There have been some studies aiming to identify renaming opportunities, e.g., Liu et al. [10], Allamanis et al. [11] and Suzuki et al. [29]. Among these approaches, the tool *RenameExpander* proposed by Liu et al. [10] is the state-of-the-art approach and achieves the best results. Hence, we employ *RenameExpander* as the baseline approach for comparison. *RenameExpander* consists of three modules, i.e., a renaming analyzer, a search engine, and a recommender. The renaming analyzer uses a background monitor (an IDE's refactoring tool, e.g., the Eclipse refactoring plugin) to capture and analyze those renaming activities of a code entity e and generate a transformation script. Then, the search engine searches code entities that are related to e and computes their similarity with the entity e based on the transformation script. If a code entity is similar enough with e, then it would be considered as a candidate for renaming. Finally, the recommender ranks these code entities based on their similarity score with e and outputs those with the highest similarity scores as renaming opportunities.

C. Evaluation Method

While preparing experimental datasets, we find that there are many identical method names within these selected Java projects, which are partially because of the widely use of overloading, overriding, and code clone in Java code. Such a phenomenon would easily cause the data leakage problem, in that once a method has been renamed, other methods with the same name would have higher probabilities to be renamed. Hence, in order to prevent data leakage within a project and to ensure the evaluation reliability of different approaches,

TABLE II
DETAILED RESULTS OF OUR APPROACH AGAINST THE BASELINE.

Project	Precision(%)		Recall(%)		F-measure(%)	
	Ours	*Baseline*	*Ours*	*Baseline*	*Ours*	*Baseline*
dubbo	90.81	**92.59**	**99.00**	89.28	**94.72**	90.90
flink	65.35	**93.38**	**98.13**	92.62	78.45	**93.00**
cassandra	73.80	**87.27**	**94.25**	47.64	**82.73**	61.63
storm	**89.86**	85.15	**91.95**	91.21	**91.74**	88.08
tomcat	**55.47**	46.23	**96.90**	49.77	**70.54**	47.94
jmeter	61.41	**75.07**	**97.28**	77.67	75.28	**76.35**
zeppelin	65.34	**83.87**	**86.13**	81.25	74.09	**82.53**
beam	62.10	**72.31**	**94.60**	74.20	**74.96**	73.24
hbase	81.45	**83.64**	**97.47**	84.03	**88.37**	83.84
camel	54.63	**80.56**	**94.76**	39.83	**69.29**	53.30

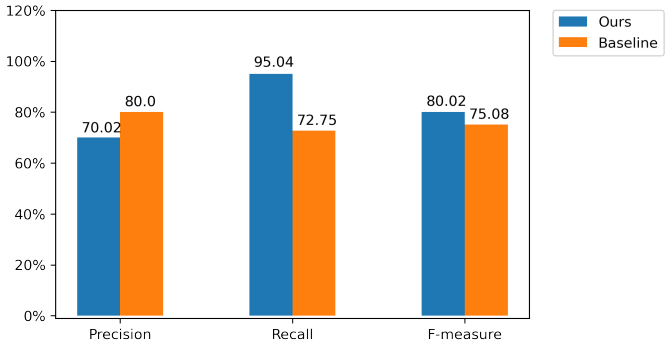

Fig. 4. Average results of our approach against the baseline.

we decided to perform cross-project prediction for method renaming. Specifically, for ten experimental projects, we use each project as the testing set while the remained nine projects as the training set for prediction model building. By repeating the model building and prediction for ten times, we take the average results of individual predictions as the final results to evaluate the performance of different approaches.

D. Evaluation Metrics

In this study, we employ three commonly-used metrics, i.e., precision, recall, and F-measure to evaluate the effectiveness of different approaches. Precision represents the ratio of relevant items (i.e., truly method renaming opportunities) among the retrieved items (i.e., potential method renaming opportunities predicted by a model), which can be computed as follows.

$$Precision = \frac{\#\ correctly\ identified\ opportunities}{\#\ predicted\ opportunities} \quad (4)$$

Recall represents the ratio of relevant items that are retrieved over all relevant items, i.e., the retrieved truly method renaming opportunities among all truly method renaming opportunities, which can be computed as follows.

$$Recall = \frac{\#\ correctly\ identified\ opportunities}{\#\ actual\ opportunities} \quad (5)$$

F-measure is a weighted summed average of precision and recall, which can be computed as follows.

$$F - measure = \frac{2\ *\ Precision\ *\ Recall}{Precision\ +\ Recall} \quad (6)$$

V. EXPERIMENTAL RESULTS

In this section, we try to evaluate the performance of our approach by answering the following Research Questions (RQs).

A. Performance Evaluation

RQ1: Can our approach outperform the state-of-the-art approach in identifying method renaming opportunities?

Motivation. Liu et al. proposed an approach of identifying method renaming opportunities, i.e., *RenameExpander* [10]. This approach is the state-of-the-art approach. Hence, we employ this approach as the baseline approach for comparison. To validate whether our approach is superior to *RenameExpander* and to what extent our approach can outperform *RenameExpander*, we set up this RQ.

Approach. Since Liu et al. do not open the source code and the experimental dataset of *RenameExpander* to the public [10], and their provided plugin of *RenameExpander* is not well applicable to a large scale of test data, we decide to implement it according to its workflow and details by ourselves. During algorithm implementation, we conduct several rounds of code review and relevant discussions to guarantee that our replication of *RenameExpander* is correct. Both our approach and *RenameExpander* are evaluated and compared against the same dataset described in Section IV.

Results. Table II shows the detailed results of our approach against *RenameExpander* and Fig. 4 shows the average comparison results, where *ours* stands for our approach and *baseline* stands for the state-of-the-art baseline approach *RenameExpander*. As shown in Table II, both of our approach and *RenameExpander* have different performance in different projects. For example, our approach achieves the precision of 90.81% in the *dubbo* project while it only achieves 54.63% in the *camel* project. Similar trends can also be observed in *RenameExpander*. The reasons behind those results maybe that different projects have different characteristics (as shown in Table I) and different approaches focus on and capture different aspects of different projects thus achieve different results.

When comparing our approach against *RenameExpander*, we can see that our approach achieves better results than *RenameExpander* in terms of recall and F-measure but underperformed in precision results. As shown in Table II, for the precision metric, despite our approach achieves not bad precision (ranging from 54.63% to 90.81%), *RenameExpander* outperforms our approach in 8 out of 10 projects, with precision being between 46.23% and 93.38%. For the recall metric, it is noteworthy that the recall of *RenameExpander* is between 39.83% and 92.62%, while that of our approach is between 86.13% and 99.00%. Our approach achieves a recall of $> 90\%$ in 9 projects and outperforms *RenameExpander* in all the 10 projects. In terms of F-measure, the F-measure of our approach on 10 projects are between 69.29% and 94.72%,

TABLE III
DETAILED RESULTS OF OUR APPROACH AND ITS VARIANT WITHOUT RELATED CODE ENTITIES.

Project	Precision(%)		Recall(%)		F-measure(%)	
	Ours	*Variant*	*Ours*	*Variant*	*Ours*	*Variant*
dubbo	**90.81**	16.51	**99.00**	**99.00**	**94.72**	28.30
flink	**65.35**	13.02	**98.13**	**98.13**	**78.45**	22.99
cassandra	**73.80**	29.88	**94.25**	**94.25**	**82.73**	45.37
storm	**89.86**	4.09	**91.95**	**91.95**	**91.74**	7.82
tomcat	**55.47**	48.66	**96.90**	**96.90**	**70.54**	64.78
jmeter	**61.41**	35.13	97.28	**97.67**	**75.28**	51.66
zeppelin	**65.34**	14.24	**86.13**	**86.13**	**74.09**	24.39
beam	**62.10**	17.99	94.60	**97.31**	**74.96**	30.37
hbase	**81.45**	69.72	**97.47**	**97.47**	**88.37**	79.94
camel	**54.63**	25.82	**94.76**	**94.76**	**69.29**	40.58

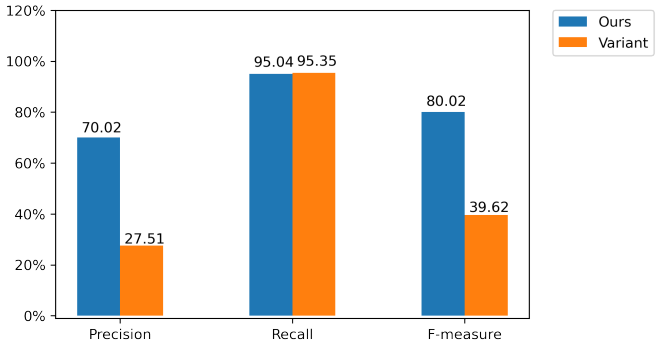

Fig. 5. Average results of our approach and its variant without related code entities.

which is generally larger than that of *RenameExpander* (with F-measure between 47.94% and 93.00%). In 7 out of 10 projects, our approach achieves better F-measure values than *RenameExpander*.

In terms of the average precision, the average precision in the 10 projects achieved by our approach (70.02%) is smaller than *RenameExpander* (80.00%). However, when considering recall, we can see that our approach achieves significantly better average recall than *RenameExpander*. On average, our approach achieves the average recall of 95.04%, while *RenameExpander* only achieves 72.75%. It means that our approach considerably outperforms *RenamExpander* by 22.29% in terms of recall. That is to say, our approach can identify renaming opportunities for methods as many as possible. In addition, our approach achieves the average F-measure of 80.02%, which is better than that of *RenameExpander* (75.08%).

Conclusion. Our approach is superior to the state-of-the-art baseline approach in terms of the average recall by 22.29% and the average F-measure by 4.94%.

B. Evaluation of Employing Related Code Entities

RQ2: How helpful are related code entities in improving the performance of our approach?

Motivation. As mentioned in section III, after we obtain the initial prediction results, we further make use of related code entities to correct and refine the potential method renaming opportunities. This RQ aims to validate the contribution of related code entities to our approach.

Approach. We develop a variant of our approach by removing the part of using related code entities for prediction results refinement. This means the variant only relies on the classifier obtained from the training phase (as shown in Fig. 2). By comparing our original approach against its variant, we can know how much employing related code entities can help to improve the performance of our approach.

Results. The detailed results of our approach and its variant are shown in Table III and the average results are presented in Fig. 5. From Table III, we can find that, the precision of our approach ranges from 54.63% to 90.81% in 10 projects, which substantially outperforms its variant whose precision ranges from 4.09% to 69.72%. In terms of recall, our approach is comparable to its variant with similar recall value in all the 10 projects (our approach achieves the same recall value with its variant in 8 out of 10 projects). In addition, our approach also achieves much better results on 10 projects than its variant in terms of the F-measure. The F-measure of our approach is between 69.29% and 94.72%, while its variant is only between 7.82% and 79.94%. These results show that our approach integrating with related code entities could largely improve the precision and F-measure but preserve similar recall with its variant.

Furthermore, from Fig. 5, we can observe that the average precision of our approach is 70.02%, while that of its variant is only 27.51%. The average recall achieved by our approach is comparable good with that of its variant. For instance, the disparity of the average recall achieved by our approach and its variant is small, i.e., 95.04% and 95.35%. In terms of F-measure, we can also find that the performance of our approach is obviously better than that of its variant. The average F-measure achieved by our approach is 80.02%, while its variant only achieves 39.62%. These results demonstrate the effectiveness of using related code entities in prediction result adjustment in our approach.

Conclusion. Employing related code entities can (without a loss of recall) effectively improve the performance of our approach in terms of the average precision and the average F-measure by 42.51% and 40.40% respectively.

C. Influence of the Size of Training Corpus

RQ3: How would the training-set scale affect the performance of our approach?

Motivation. Our approach relies on a training set to identify renaming opportunities. To understand how exactly the training-set scale would affect the performance of our approach, we set up this RQ.

Approach. As mentioned in Section IV-C, we conduct cross-project prediction in identifying method renaming opportunities, with 9 projects as the training set and the remaining one as the test set. To answer this RQ, we changed the sizes of the training set by only including the first 1, 3, 5, 7, 9 projects (ordered by the *star* number) and taking each project as the test set. If the test project happens to be in the first k

TABLE IV
DETAILED RESULTS OF OUR APPROACH WITH DIFFERENT SIZES OF THE TRAINING CORPUS.

Project	Precision(%)					Recall(%)					F-measure(%)				
	K=1	*K=3*	*K=5*	*K=7*	*K=9*	*K=1*	*K=3*	*K=5*	*K=7*	*K=9*	*K=1*	*K=3*	*K=5*	*K=7*	*K=9*
dubbo	**93.09**	90.85	90.05	90.72	90.81	70.50	**99.50**	94.00	98.00	99.00	79.54	91.44	**94.97**	94.19	94.72
flink	**72.05**	65.34	62.25	65.64	65.35	75.58	98.03	97.64	**99.31**	98.13	72.29	78.41	78.22	**79.04**	78.45
cassandra	**82.88**	76.64	73.09	73.57	73.80	63.37	91.96	93.37	90.94	**94.25**	71.67	82.33	81.95	81.27	**82.73**
storm	25.94	12.44	9.24	87.32	**89.86**	84.82	76.49	47.12	57.74	**91.95**	39.65	21.41	13.55	67.33	**91.74**
tomcat	54.34	**60.65**	55.40	55.32	55.47	72.33	76.28	**98.12**	96.40	96.90	62.01	66.33	**70.82**	70.32	70.54
jmeter	76.37	**78.80**	61.84	61.55	61.41	56.93	62.74	**99.29**	97.63	97.28	64.36	68.56	**76.22**	75.48	75.28
zeppelin	13.60	13.60	38.71	36.86	**65.34**	79.68	79.68	73.37	**86.13**	82.13	24.30	23.17	50.63	47.60	**74.09**
beam	67.90	**69.47**	62.11	61.80	62.10	78.44	64.97	**95.55**	94.36	94.60	72.48	66.28	**75.28**	74.68	74.96
hbase	74.87	57.73	77.38	78.23	**81.45**	77.39	71.06	81.27	94.66	**97.47**	75.75	63.01	78.40	85.09	**88.37**
camel	60.85	**62.73**	55.54	55.15	54.63	80.81	82.60	**98.14**	94.23	94.76	69.29	70.26	**70.93**	70.80	69.47

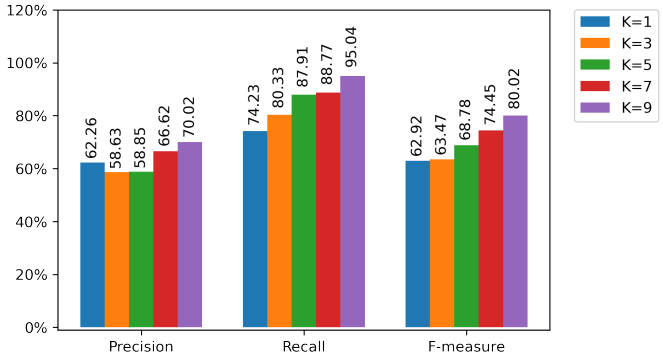

Fig. 6. Average results of our approach with different sizes of the training corpus.

training projects, we then modify the training set by replacing the project with its subsequent project. All projects are used individually as the test set and we take the average results as the final performance results to investigate this RQ.

Results. Table IV and Fig. 6 show the detailed and average results of our approach with different sizes of the training set. We can see that along with the increase of the training-set size, the performance of our approach shows upward trends on the whole. For example, in the *hbase* project, our approach achieves the best results in terms of precision, recall, and F-measure, when 9 projects are used as the training set. Note that there are still some exceptions which do not absolutely follow such a trend. As shown in Table IV, compared with large training sets (k is large), our approach in the *camel* project can also achieve comparable results when only one project is used in the training set. For example, when k=1, our approach achieves the F-measure of 69.29% in the *camel* project. In contrast, when k=9, our approach achieves comparable F-measure of 69.47%. Similarly, our approach can achieve comparable or even the best results in the *dubbo, flink, tomcat, jmeter, beam* projects, when there are fewer than 9 projects working as training sets. It means that even with relatively small training sets, our approach can also achieve satisfactory results in some projects.

In terms of the average results shown in Fig. 6, our approach still shows upward trends on the whole along with the increase of the training-set scale. For example, when there is only one project in the training set (k=1), our approach achieves the average precision, recall, and F-measure of 62.26%, 74.23%, and 62.92%. When k increases to 5, the three evaluation metrics achieved by our approach increase to 58.85%, 87.91%, and 68.78%. In addition, when we regard the rest 9 projects as the training set (k=9), our approach achieves the average precision, recall, and F-measure of 70.02%, 95.04%, and 80.02%.

Conclusion. On the whole, our approach generally performs better along with the increase of the training-set scale. Meanwhile, our approach can still achieve satisfactory results when the size of the training set is relatively small in some projects.

VI. THREATS TO VALIDITY

In this section, we discuss the threats to validity in the approach, including threats to internal validity and threats to external validity.

Threats to Internal Validity. In this paper, we assume that there are functional overlapping relationships between related code entities (as shown in Fig. 1, the method name *getMaxInteger* and invoked method *println* are related entities that have functional relationships). The correction process of our approach heavily relies on related code entities, i.e., a method is identified to be renamed only if its related code entities have been renamed. However, this assumption may be inaccurate in practice, which may threaten the validity of our approach.

Threats to External Validity. In this paper, we validate our approach on 10 open source Java projects with 114,398 methods. We cannot guarantee that our conclusions could be applicable to other projects. However, considering that the selected 10 projects are all popular projects that are well-maintained, we believe that our observations could still provide some hints on the effectiveness of our approach in practice. We plan to further alleviate this threat by validating our approach on more diverse projects in the future.

VII. RELATED WORK

There have been a number of studies on code entity renaming. Most of these studies predict renaming by investigating the constitutive tokens, their orders, and their types in code

entities. Caprile and Tonella [30], [31] proposed a lexicon-based approach that identified code entities not in standard dictionaries (e.g., keywords of programming languages) as renaming opportunities. Reiss [32], Allamanis et al. [33], and Suzuki et al. [29] proposed the statistical language model to model naming conventions and used probabilistic evaluation to identify renaming opportunities.

To address the limitation of lexical information and take advantage of the machine learning technique, some researchers recently tried to integrate semantic features of code entities to better identify code-entity renaming opportunities. Allamanis et al. [11] proposed a log-linear context model, in which field vectors with similar semantics would be assigned with similar positions and those field vectors not similar to their contexts would be taken as renaming candidates. Anh et al. [12] proposed a statistical model *APIREC*, which captured frequent fine-grained code changes to infer the next code change for renaming. Liu et al. [10] proposed an approach to extend historical renaming to identify renaming opportunities. Liu et al. [13] used paragraph vectors and textCNN to extract semantic features of method names and bodies to identify their inconsistency as renaming opportunities.

Our approach is different from those approaches in the following two aspects. On one hand, we employed BERT to capture the overlapping relationships of code entities to better represent the semantics of methods. On the other hand, we leveraged historical renaming activities of related code entities to refine the classification results for method renaming identification. This makes our approach greatly outperform the state-of-the-art approach in method renaming identification.

VIII. Conclusion and Future work

As an important type of code refactoring, renaming plays a key role in program comprehension and software defect detection. However, identifying renaming opportunities still remains a challenging research task, especially for methods. In this study, we propose a novel approach to identify method renaming opportunities by fully leveraging historical code changes and overlapping relationships among code entities. Experimental results on 10 open source Java projects with a total of 114,398 methods show that our proposed approach achieves an average F-measure of 80.02% in identifying renaming opportunities and improves the state-of-the-art approach by 4.94% on average.

In the future, we plan to validate our approach on projects from diverse domains and with different scales. We also plan to extend our approach to identify renaming opportunities for other code entities such as field names and local variables.

Acknowledgment

This work is supported by the National Natural Science Foundation of China under Grant No. 61902181 and 62002161, the China Postdoctoral Science Foundation under Grant No. 2020M671489, the CCF-Tencent Open Research Fund under Grant No. RAGR20200106, and the Nanjing University of Aeronautics and Astronautics Postgraduate Research and Practice Innovation Program under Grant No. xcxjh20211612.

References

[1] F. Deißenböck and M. Pizka, "Concise and consistent naming," *Software Quality Journal*, vol. 14, pp. 261–282, 2005.

[2] L. M. Eshkevari, V. Arnaoudova, M. Penta, R. Oliveto, and G. Antoniol, "An exploratory study of identifier renamings," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, 2011, pp. 33–42.

[3] D. Shepherd, L. Pollock, and K. Vijay Shanker, "Case study: supplementing program analysis with natural language analysis to improve a reverse engineering task," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2007, pp. 49–54.

[4] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, , "Exploring the influence of identifier names on code quality: An empirical study," in *Proceedings of the 14th European Conference on Software Maintenance and Reengineering*. IEEE, 2010, pp. 156–165.

[5] V. Arnaoudova, L. Eshkevari, M. D. Penta, R. Oliveto, G. Antoniol, and Y. G. Guéhéneuc, "Repent: Analyzing the nature of identifier renamings," *IEEE Transactions on Software Engineering*, vol. 40, pp. 502–532, 2014.

[6] A. Schankin, A. Berger, D. V. Holt, J. C. Hofmeister, T. Riedel, and M. Beigl, "Descriptive compound identifier names improve source code comprehension," in *Proceedings of the 2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 2018, pp. 31–3109.

[7] L. Ben, B. Andrew, and S. Eve, "Cognitive perspectives on the role of naming in computer programs," in *Proceedings of the 18th Annual Workshop of the Psychology of Programming Interest Group*, p. 11.

[8] G. Li, H. Liu, and A. S. Nyamawe, "A survey on renamings of software entities," *ACM Computing Surveys*, vol. 53, pp. 1–38, 2020.

[9] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Transactions Software Engineering*, vol. 28, pp. 970–983, 2002.

[10] H. Liu, Q. Liu, Y. Liu, and Z. Wang, "Identifying renaming opportunities by expanding conducted rename refactorings," *IEEE Transactions on Software Engineering*, vol. 41, no. 9, pp. 887–900, 2015.

[11] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 38–49.

[12] A. T. Nguyen, M. Hilton, M. Codoban, H. A. Nguyen, L. Mast, E. Rademacher, T. N. Nguyen, and D. Dig, "Api code recommendation using statistical learning from fine-grained changes," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 511–522.

[13] K. Liu, D. Kim, T. F. Bissyandé, T. Kim, K. Kim, A. Koyuncu, S. Kim, and Y. Le Traon, "Learning to spot and refactor inconsistent method names," in *Proceedings of the 41st International Conference on Software Engineering*, 2019, pp. 1–12.

[14] R. Takanobu, T. Zhang, J. Liu, and M. Huang, "A hierarchical framework for relation extraction with reinforcement learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2019, pp. 7072–7079.

[15] J. Devlin, M. W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2019, pp. 4171–4186.

[16] Y. Kim, "Convolutional neural networks for sentence classification," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, 2014, pp. 1746–1751.

[17] R. L. Russell, L. Y. Kim, L. H. Hamilton, T. Lazovich, J. A. Harer, O. Ozdemir, P. M. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," *Proceedings of the 17th IEEE International Conference on Machine Learning and Applications*, pp. 757–762, 2018.

[18] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing*, pp. 3980–3990, 2019.

[19] P. Xia, S. Wu, and B. Van Durme, "Which *bert? A survey organizing contextualized encoders," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*, 2020, pp. 7516–7533.

[20] B. Li, H. Zhou, J. He, M. Wang, Y. Yang, and L. Li, "On the sentence embeddings from pre-trained language models," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*, 2020, pp. 9119–9130.

[21] G. E. Dahl, R. P. Adams, and H. Larochelle, "Training restricted boltzmann machines on word observations," in *Proceedings of the 29th International Coference on International Conference on Machine Learning*, 2012, pp. 1163—1170.

[22] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[23] T. Mikolov, G. Corrado, C. Kai, and J. Dean, "Efficient estimation of word representations in vector space," in *1st International Conference on Learning Representations, ICLR, , Workshop Track Proceedings*, 2013.

[24] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[25] X. Shi, Z. Chen, H. Wang, D. Y. Yeung, W. K. Wong, and W. C. Woo, "Convolutional lstm network: A machine learning approach for precipitation nowcasting," in *Advances in neural information processing systems*, 2015, pp. 802–810.

[26] Z. Dai, Z. Yang, Y. Yang, J. Carbonell, Q. Le, and R. Salakhutdinov, "Transformer-XL: Attentive language models beyond a fixed-length context," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019, pp. 2978–2988.

[27] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, 2016, pp. 1715–1725.

[28] A. Sheneamer and J. Kalita, "Code clone detection using coarse and fine-grained hybrid approaches," in *2015 IEEE Seventh International Conference on Intelligent Computing and Information Systems*, 2015, pp. 472–480.

[29] T. Suzuki, K. Sakamoto, F. Ishikawa, and S. Honiden, "An approach for evaluating and suggesting method names using n-gram models," in *Proceedings of the 22nd International Conference on Program Comprehension*, 2014, pp. 271–274.

[30] B. Caprile and P. Tonella, "Restructuring program identifier names," in *Proceedings 2000 International Conference on Software Maintenance*, 2000, pp. 97–107.

[31] B. Caprile and P. Tonella, "Nomen est omen: Analyzing the language of function identifiers," in *Proceedings of the Sixth Working Conference on Reverse Engineering*, 1999, pp. 112–122.

[32] S. P. Reiss, "Automatic code stylizing," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, 2007, pp. 74–83.

[33] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 281–293.