IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received November 14, 2019, accepted December 13, 2019, date of publication January 14, 2020, date of current version February 10, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2966532

Review Sharing via Deep Semi-Supervised Code Clone Detection

CHENKAI GUO[1], (Member, IEEE), HUI YANG[1], DENGRONG HUANG[1], (Member, IEEE), JIANWEN ZHANG[3], NAIPENG DONG[2], JING XU[3], (Member, IEEE), AND JINGWEN ZHU[4]

[1]College of Computer Science, Nankai University, Tianjin 300350, China
[2]School of Computing, National University of Singapore, Singapore
[3]College of Artificial Intelligence, Nankai University, Tianjin 300350, China
[4]College of Software, Nankai University, Tianjin 300350, China

Corresponding author: Jing Xu (xujing@nankai.edu.cn)

This work was supported in part by the Science and Technology Planning Project of Tianjin, China, under Grant 17JCZDJC30700, Grant 17YFZCGX00610, and Grant 18ZXZNGX00310, in part by the Tianjin Natural Science Foundation under Grant 19JCQNJC00300, and in part by Fundamental Research Funds for Central Universities of Nankai University under Grant 63191402.

ABSTRACT Code review as a typical type of user feedback has recently drawn increasing attentions for improving code quality. To carry out research on code review, sufficient review data is normally required. As a result, recent efforts commonly focus on analysis for projects with sufficient reviews (called ''s-projects''), rather than projects with extremely few ones (called ''f-projects''). Actually, through statistics on public platforms, the latter ones dominate open source software, in which novel approaches should be explored to improve their review-based code improvement. In this paper, we try to address the problem via building a review sharing channel where the informative review can be reasonably delivered from s-projects to the f-projects. To ensure the accuracy of shared reviews, we introduce a novel code clone detection model based on Convolutional Neural Network (CNN), and build suitable ''s-projects, f-projects'' pairs through the clone detection. Especially, to alleviate the dataset heterogeneity between the training and testing, an autoencoder-based semi-supervised learning strategy is employed. Furthermore, to improve the sharing experience, heuristic filtering tactics are applied to reduce the time cost. Meanwhile, the LDA (Latent Dirichlet Allocation)-based ranking algorithm is used for presenting diverse review themes. We have implemented the sharing channel as a prototype system RSharer+, which contains three representative modules: data preprocessing, code clone detection and review presentation. The collected datasets are first transformed into context-sensitive numerical vectors in the data proprecessing. Then in the clone detection, data vectors are trained and tested on the BigCloneBench and real code-review pairs. At last, the presentation module provides review classification and theme extraction for better sharing experience. Extensive comparative experiments on hundreds of real labelled code fragments demonstrate the precision of clone detection and the effectiveness of review sharing.

INDEX TERMS Code clone, software review, deep learning, semi-supervised CNN, review sharing.

I. INTRODUCTION

Code review as typical type of user feedback has drawn increasing attentions in the research field of code improvement lately. Many research topics aiming at how to effectively leverage the review resources have been studied, such as review recommendation [1]–[3], review quality evaluation [4], [5], review localization [6], [7] and review classification [8], [9]. When focusing on the review-based applications, researchers are prone to overlooking a latent but critical threat: not all software projects have sufficient reviews. Actually, according to statistics on 4 popular code management platforms (from Jan 2018 to May 2019) (Figure 1), the s-projects (software projects with more than 50 reviews) only account for 17.2%, while the f-projects (projects with fewer than 10 reviews) account for 3 time more. The lack of reviews for the majority of code projects seriously limit the usability and effectiveness of existing review-based code analysis. To address the problem, some review generation solutions have been proposed recently. For instance, Xia et al. [10], [11] proposed DeepCom, which adopts deep encoder-decoder mechanism to automatically generate code comments for Java

The associate editor coordinating the review of this manuscript and approving it for publication was Xiaobing Sun.

methods. Haije *et al.* [12] leverage sequence-to-sequence deep neural network to translate the code into corresponding reviews. Zheng *et al.* [13] and Iyer *et al.* [14] [13] and [14] introduce the attention mechanism into the code-based comment generation, in which the deep encoder-decoder and some domain features are correspondingly used. However, such generative models are hard to construct human-like smooth review sentences. For example, the review sentence ''How baseUrl emailId C how to a page inBeginGetResponse to'' generated by the proposed tool in [14] is hard to be understood in that they contain grammatical and spelling errors. Generally, the generated reviews just contain some keywords, yet are hard to be completely understood. From this perspective, sharing suitable human-submitted reviews from existing s-projects to the f-projects can be a feasible choice. Along with the direction, Wong *et al.* [15] has proposed a sharing strategy relying on the technique of code clone detection and has achieved some reasonable comment sharing results. Nevertheless, the limited capability of the clone detection and coarse-grained review ranking strategy negatively impact the sharing effectiveness. Our previous work [16] has proposed to introduce the word embedding in NLP (natural language processing) and CNN (convolutional neural network) model to improve the coverage and precision of clone detection. Meanwhile, several heuristic clone search strategies are adopted to enhance the sharing performance. However, there are still two-fold of challenges. The first challenge is that the clone detection model is trained and tested on the datasets with different code sources. The training dataset is a common-used Java code clone benchmark BigCloneBench [17]; while the testing dataset is the real collected code fragments from open source platforms. Such difference leads to evident detection deviation. The second challenge lies in the presentation for sharing results. Both of [15] and [16] adopt simple similarity-relevant ranking and presentation tactics, which is hardly meet developer's requirements.

This paper is an extension of our previous work [16]. The improvement targets at aforementioned two-fold of challenges. For the first challenge, we integrate an autoencoder-based semi-supervised module into the CNN detection model to enhance the detection precision for the code fragments without clone labels. For the second challenge, we introduce review classification as well as extract useful review themes by LDA (Latent Dirichlet allocation) for better representation for developers. In details, we present a systematic review sharing framework RSharer+ in this paper. The RSharer+ contains three core modules: data preprocessing, code clone detection and sharing presentation. In data preprocessing, the labelled and unlabelled datasets are collected and transformed into context-sensitive numerical vectors. For code clone detection, the autoencoder-based semi-supervised CNN model is proposed to alleviate the dataset heterogeneity, which is trained on the BigCloneBench and real code-review pairs. For sharing presentation, the informative reviews are automatically picked and ranked through similarity computation and review classification. Meanwhile, the LDA model is employed to assign the reviews with meaningful themes. Extensive comparative experiments on hundreds of real labelled code fragments illustrate the precision of clone detection and the effectiveness of review sharing.

Overall, the contribution of this work can be summarised as follows:

1) We propose a novel clone-based review sharing approach to resolve the lack-of-reviews problem within review-based code analysis. Especially, an autoencoder-based semi-supervised deep learning technique is employed to alleviate the dataset heterogeneity of clone detection.

2) We integrate techniques of review classification and theme extraction into the review sharing for better sharing presentation.

3) We implemented a prototype system RSharer+ to validate the proposed sharing approach. Extensive comparative experiments on hundreds of real code fragments are conducted to verify the precision of clone detection and effectiveness of sharing presentation.

The remaining parts of this paper are organised as follows. We introduce the motivation of the work using a real world example in Section II, and discuss the related work in Section III. The proposed approach, as well as the implementation details, is presented in Section IV. In the subsequent Section V, we present the experiment results and answer the research questions. In particular, the subsection V-E discusses some threats towards validity. Finally, Section VI concludes the entire work.

II. MOTIVATION

The motivation of this work can be demonstrated using the code fragment shown in Figure 2, which contains two projects captured from Github sharing the same code fragment. In the figure, the left hand side is the shared code fragment which implements a *WordCounter* class; and the right hand side are the two projects (*project A* [1] and *project B*)[2] that contains the code fragment on the left hand side. As illustrated, the *project A* does not have any review; while the *project B* receives many reviews, some of which (e.g., the reviews in red) are closely related to the shared code fragment. It is well-known that with the reviews, it much easier for the developers of *project B* to identify the problem in the shared code fragment, while without any review, it is challenging for the developers of *project A* to find such problem. This inspires us to explore whether there is a way to assist those ''developers of *project A*'' to obtain valuable reviews by learning from reviews of *project B*. In this paper, we carry out the a systematical attempt towards this goal.

III. RELATED WORK

A. CODE CLONE

Code clones refer to two or more code fragments that belong to a clone type [18]. There has been a lot of efforts on

[1] From https://github.com/paulpaulych/WordCounter

[2] From https://github.com/exercism/java

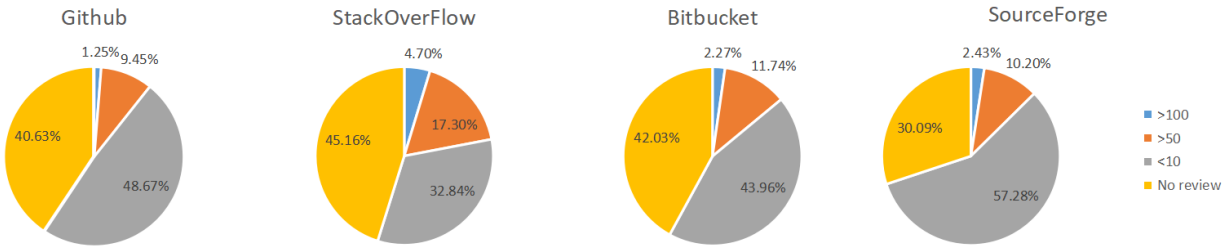

FIGURE 1. Statistics for software reviews.

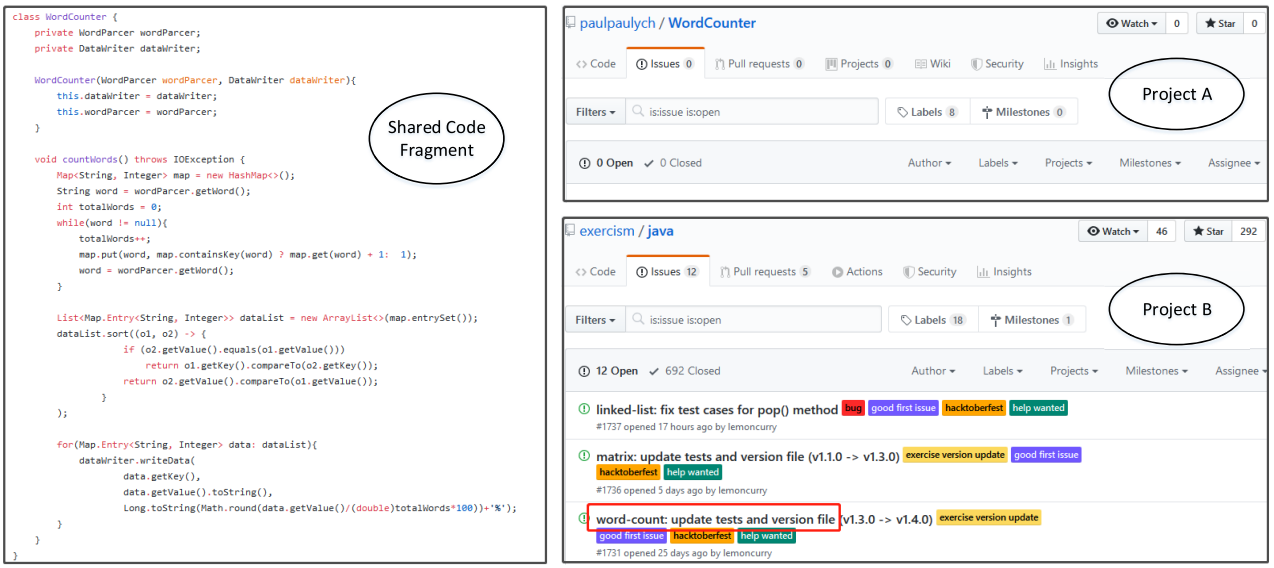

FIGURE 2. Motivation example.

code clone detection. Depending on the exploited source code representation, there are in general four types of code clone detection, namely text-based, token-based, tree-based and graph-based [19].

Text-based techniques [20]–[24] detect code clone by measuring the similarity of two pieces of code, which is quantified by comparing the order of texts. A well-known work is *dup* [22] which checks duplicated or related code sequences in larger software systems. The downside of text-based techniques is its limited ability to recognize cloned pairs at fragment levels. It even fails to detect the clone when the differences between two fragments are insignificant, e.g., the systematic renaming of the identifier.

Token-based techniques [25]–[29] add a token analysis stage, which analyses the code in lexical level to produce a stream of tokens. Then based on the token sequences, similarity between two code fragments is calculated. A typical example is CCFinder [27] by Kamiya *et al.*, which generates tokens for each code line according to given lexical rules. In addition, Wang et al. developed a token-based clone detector - CCAligner [30], targeting at large-gap clone detection. Due to token analysis, token-based approaches partially address the code variation limitation in text-based clone detection. And thus they are able to handle type-2 clones (IV-C).

Tree-based techniques [31], [32], [32]–[34] represent code fragments as ASTs (abstract syntax trees) or other tree-like structures, instead of stream of tokens. Then the similarity of subtrees is measured in syntactic representation. One of the most representative works is Deckard [33], where the parse trees are represented using vectors and the similarity of vectors is used for clone detection. While Koschke et al. use suffix trees to detect syntactic clones, which is linear in time and space [32]. In recent years, CClearner [34] and other deep learning based models [35] introduce deep learning into tree-based techniques and achieved good results. However, tree-based techniques lead to a time-consuming and space-consuming dilemma [29].

Graph-based techniques use static program analysis [36]–[41] to convert code into graphical representations e.g., program dependence graph (PDG) [42] and control flow graph (CFG) [43]. For instance, Gabel *et al.* [36] proposed a scalable clone detection algorithm, which transforms the difficult graph-similarity-calculation problem into a simpler tree-similarity-calculation problem by mapping carefully selected PDG subgraphs to their corresponding structured

syntax. Liu et al. [39] implemented GPLAG, a tool that detects plagiarism by mining programs that rely on graphs. Chen *et al.* [40] detect code clones in two apps by combining CFG and PDG to calculate ''geometry characteristic''. Compared to the above mentioned techniques, graph-based techniques perform better in near-miss clone detection thanks to the high-level abstraction and taking semantics into consideration. The downside of this approach is that building the graph structures is usually expensive [34].

Different from existing clone detection works, our learning-based approach employs semi-supervised CNN, which not only well performs for the type-3 and type-4 clones (IV-C), but also resolves the data heterogeneity between training and testing.

B. INFORMATIVE TEXT LINKING WITH SOURCE CODE

Code review as a typical type of feedback include various information, such as bug reports, user experience, etc. Therefore, linking the informative text with its related source code is an essential task, as these informative text reviews can help improve the code quality, such as locating bug occurrence, improving usage experience, etc. Researchers have proposed related detection approaches [44]–[48], [48]. For example, Palomba et al. present CHANGEADVISOR [6], a link detector considering the structure, semantics, and sentiments of sentences contained in user reviews. The text code links have been proposed to detect bug localization in the works [49], [6], [45] and [50]. As an example, Saha et al. developed BLUiR [49], a practical link detector using the Vector Space Model [51] to link bug reports and source code. Besides these works, Di Sorbo *et al.* [44], [46] and Bacchelli et al. [45] have presented benchmarks and approaches to link email and code. In particular, Di Sorbo et al. [44], [46] exploit the use of semi-supervised learning methods in finding such links.

In this paper, the pairs of informative review and corresponding code fragment are collected as Ground Knowledge (V-A.2). The reviews are shared according to the similarity measurement of code fragments.

C. DEEP LEARNING IN SOURCE CODE ANALYSIS

Recently, some deep learning methods have been used to solve problems in source code analysis [19], [52]–[55]. Some researchers focus on the application of deep learning in code completion and code generation. White *et al.* [52] attempted to introduce deep learning into the field of software language modeling. Tony Beltramelli [53] implemented pix2code, a method based on CNN and RNN to automatically generate computer code from a single GUI screenshot as input. Specially, after pix2code, Chen *et al.* [55] implemented ui2code, a kind of neural machine translator, which combines computer vision and machine translation to translate a UI design image into a GUI skeleton. Mou *et al.* [54] proposed an encoder-decoder model based on RNN to convert the intention expressed in natural language into code. Other researchers have explored deep learning for code clone detection. Wei and Li [56] proposed an end-to-end deep feature learning framework called CDLH based on LSTM and hash translation, which can be used to functional clone detection. Li *et al.* [34] presented CCLEARNER, which is the first solely token-based clone detection using deep learning. CCLEARNER first extracts tokens from both known method-level code clones and non-clones, which will be used to train a classifier, and then the classifier is used to detect clones in a provided code database. In addition, there are also researchers explore deep code search [57], API mapping [58], [59], code summarization [60], [61] and bug localization [62]. Among them, Lam *et al.* [62] build HyLoc, a model combining revised Vector Space Model (rVSM) with DNN to recommend the potentially buggy files for a bug report. In HyLoc, DNN is used to measure the relevancy between buggy files and bug report. Essentially, DNN learns the correspondence between the terms in bug reports, as well as the potential code tokens, with terms in source files.

Inspired by existing works, in this paper, we aim to use deep learning techniques to establish a review sharing channel for improving the quality of code projects.

IV. APPROACH

A. OVERVIEW

The structure of proposed review sharing framework RSharer+ is demonstrated as Figure 3, which contains three main modules: data preprocessing, clone detection and sharing presentation. In data preprocessing, the code and reviews are transformed into context-sensitive numerical vectors that support further deep learning-based prediction and classification. To achieve the goal, we first collect sufficient code pairs from both of the code clone datasets (BigCloneBench) and code management platforms (Ground Knowledge) for model training, and then automatically tag partial code pairs in the clone datasets with three types of labels, i.e., class 1 clone, class 2 clone and non-clone. Then, to preserve information of syntax and semantics, original code fragments (both from clone datasets and management platforms) are transformed into Abstract Syntax Tree (AST) (step ①) tokens. Afterwards, the popular word embedding method Word2Vec (step ②) is exploited to translate both the code AST tokens and reviews into context-sensitive vector representations. After data preprocessing, in the clone detection phase, pairs of produced vector representations (containing both of labelled and unlabelled pairs) are fed into the semi-supervised CNN module to train a multi-classification model, where the loss function is computed by the combination of MSE (mean square error) and encode-decode loss. The trained model is then used to precisely categorise a targeted pair of code vectors into three classes: class-1 clone, class-2 clone and non-clone. In addition, based on the obtained trained model, a final review ranking is computed in the sharing presentation phase. Meanwhile, to better present the shared information, the reviews are classified as four semantic-related categories and tagged as different themes.

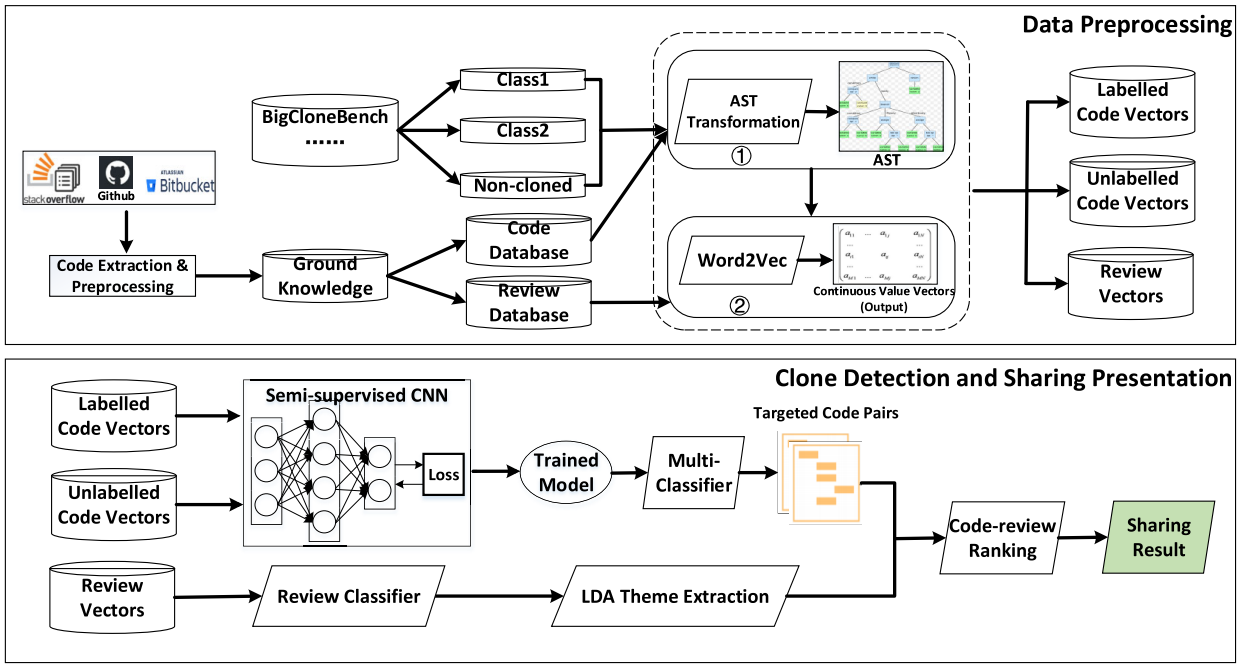

FIGURE 3. Approach overview.

For a targeted pair of code fragment vectors, the RSharer+ feeds it into the semi-supervised model that has finely trained in the second phase and obtains the detection result that can be used to determine similarity of the two code fragments. Then the reviews are shared to the target code fragment according to the similarities of different code pairs in the Ground Knowledge (V-A.2). Eventually, the targeted code fragment obtains the shared informative reviews and its corresponding themes using the review classification and LDA model.

B. DATA PREPROCESSING

1) AST TRANSFORMATION

The input datasets is collected from two sources: code clone datasets (BigCloneBench) and real code management platforms (Ground Knowledge). The code fragments from BigCloneBench are divided into three typical categories: class1 clone, class2 clone and non-cloned (introduced in details in Section V-A.1). The Ground Knowledge contains numerous code-review pairs which can be divided into code dataset and review dataset respectively.

To generate proper input vectors while preserving effective information (e.g., lexical tokens, syntactic structures and semantic information), RSharer+ first transforms the code fragments from both of the BigCloneBench and Ground Knowledge into AST - a typical representation of code structure, and treats the AST nodes as the input of further vectoring, since the AST nodes contain rich representative information for model training [63]. In practice, we extract method identifiers, qualified names, variable identifiers, literals, reserved words, operators, makers and type identifiers in the AST. Note that not all the AST nodes are used, e.g., the punctuation symbols are discarded since they do not carry much information and would increase the training cost.

Figure 4 shows a typical AST of a source code fragment, where the left hand side is the code fragment in the motivation example and the right hand side is its corresponding AST.

2) PRE-TRAINING

The input of the clone detection module (described in Section IV-C) is numeric vectors. The ideal input vectors for training need to be dense continuous values. Thus, a special pre-training phase is designed to obtain such ideal input vectors from the AST nodes. We adopt the popular Word2Vec (a typical unsupervised word embedding technique) [64] to carry out the pre-training, as it can convert code tokens into a unique real-value vector space, while preserving their potential semantic and syntactic information. This feature of Word2Vec ensures that two code fragments with similar semantics will be close to each other in the embedding space.

In practice, we use Skip-gram [65] to implement the Word2Vec, which generally outperforms other implementations in keeping contextual semantics. In more details, for a given key word w, Skip-gram model predicts the emerging possibilities of its context w_O. The training loss is defined as the *cross entropy* between the prediction context and ground truth, since lower cross entropy indicates higher similarity between two distributions. The true label y_i is 1 only when w_i is the real output word; and otherwise y_j is 0. Aiming at minimising the *cross entropy*, the loss function $\mathcal{L}_\theta$ of the model, is defined as follows, where θ is a parameter:

$$\mathcal{L}_\theta = -\sum_{i=1}^{V} y_i \log p(w_i|w_I) = -\log p(w_O|w_I) \qquad (1)$$

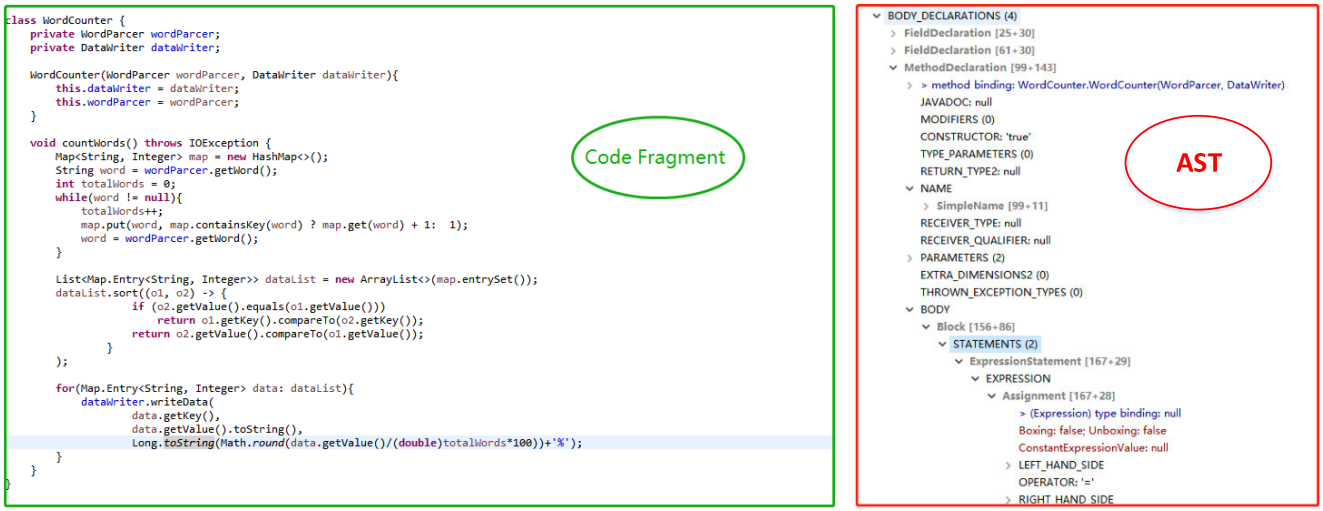

FIGURE 4. AST example.

where w_O denotes the set of output words; w_I denotes the given key words set; V is the context range.

To compute the prediction probability, the final output layer of the Skip-gram applies the probability computation of *softmax* defined as follows:

$$p(w_O|w_I) = \frac{exp({v'_{w_O}}^{\top} \cdot v_{w_I})}{\sum_{i=1}^{V} exp({v'_{w_i}}^{\top} \cdot v_{w_I})} \tag{2}$$

where v_{w_I} (embedding vector) is the row of input matrix W and v'_{w_O} (context vector) is the column of output matrix W'. Finally,

$$\mathcal{L}_\theta = -{v'_{w_i}}^{\top} \cdot v_{w_I} + \log \sum_{i=1}^{V} exp({v'_{w_i}}^{\top} \cdot v_{w_I}) \tag{3}$$

Note that, for the review dataset from Ground Knowledge, RSharer+ directly preprocesses them through the Word2Vec module.

C. CLONE DETECTION (SEMI-SUPERVISED CNN)

The code clone can be classified as four typical types, according to a classical code clone categorization methodology [17], [34]. Type-1 clones are identical code fragments except variations on whitespace, layouts, and comments. Type-2 clones are code fragments that may have variations in identifiers, literals, types, whitespace, layouts, and comments. Type-3 clones allow extra variations such as changed, added, or removed statements. Type-4 clones further allow semantically equivalent but syntactically different code fragments. Traditional clone detection approaches [29], [33] are effective in detecting the type-1 and type-2 clones, but ineffective in detecting type-3 and type-4 clones. However, in review sharing, detecting the pairs of code fragments that have the same meaning but different representations can significantly improve review sharing. And thus detecting type-3 and type-4 is essential in this work. Lately, applying the novel deep learning techniques in clone detection is becoming popular and some have achieved fairly good detection results, especially in detecting type-3 and type-4 clones. Under the insight, we tried the state-of-the-art learning-based clone detection techniques [19], [34], [66] in our work. However, two of them [34], [66] perform poor in detecting type-3 and type-4 clones in review sharing in our experimental investigation. We found that the bad performance is caused by the hand-crafted code features as well as the fact that they did not take into account of the code structure. Although the other model [19] performs better in the detection precision, it has poor recall rate. Therefore, we design a new clone detection approach, that is suitable for our review sharing goal, to further increase the detection accuracy, especially for type-3 and type-4 clone detection. Note that we treat the clone detection as multi-classification rather than regression problem, since the numerical labels for regression are difficult to obtain.

Correspondingly, we design a specific semi-supervised CNN model for the code clone detection, which makes full use of both the labelled and unlabelled code pairs. Figure 5 presents the overall architecture of the semi-supervised CNN, which contains an ordinary CNN model for labelled similarity prediction and a denoise autoencoder model for unlabelled feature learning. The approach is under the intuition that the encoder of denoise model and the CNN of clone prediction model can share the same CNN parameters, since a decent CNN feature extraction improves both of the denoise and the clone detection. In details, the autoencoder structure is inspired by the denoise autoencoder (DAE [67]), which computes two directions of mappings: an encode mapping f and a decode mapping g, and each of them consists of several CNN layers.

1) ORDINARY CNN

Inspired by classical TextCNN [68], the input of CNN module is the concatenation of a pair of AST vectors produced by preprocessing, extracted from both the targeted code and the peer

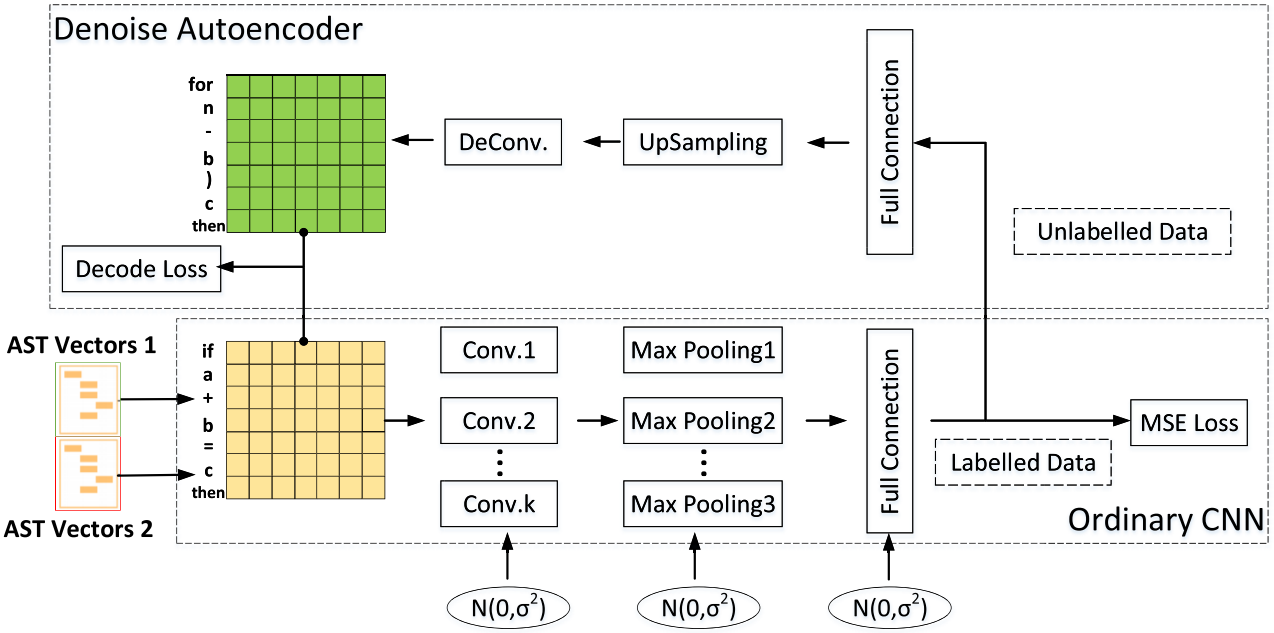

FIGURE 5. Semi-supervised CNN structure of sharer+.

code obtained from Ground Knowledge. Note that each AST node is a d-dimensional vector. We then perform convolution on the concatenated node vectors via linear filters, which is defined as follows:

$$o = k^{\top} \cdot x + b \tag{4}$$

where $k^{\top}$ is the filter kernel; x is the input word vectors; o is the output value; b is the bias item. The dot $\cdot$ refers to the dot product between the filter kernel and word vectors. In the convolution, we used filters of 5 window sizes: window size 1, window size 3, widow size 5, window size 7, window size 9 respectively, which ensures the captured features from different n-grams. And we use 200 filters For each window size to carry out the feature learning. In this process, each filter is a $d \times n$ vector (n is the window size), which preserves the entire semantics of the original target word. Then we use the non-linear activation function *ReLu* to transform each of output value into a feature map, where $c_i = ReLu(o_i)$. Note that the size of the obtained feature map varies along with the size of the filter and the input vectors. Therefore, the 1-max pooling is used to extract a length 1 feature vector with the maximum value in the feature maps for each filter. The output feature maps are finally concatenated into a single dimensional vector via a full connection layer, and then mapped into a single score which indicates the similarity between the pair of the code fragments. At last, the computed score from the last layer of CNN is used to compute the bias, together with the baseline score, according to the MSE (mean square error) loss function:

$$\mathcal{L}_p = \frac{1}{|P_l|} \sum_{i \in P_l} (Baseline_i - \tilde{y}_i)^2 \tag{5}$$

where P_l is the entire labelled code pairs; $Baseline_i$ is the baseline score for the i-th code pair; and $\tilde{y}_i$ is the prediction score for the i-th code pair.

Actually, the feature maps will be fed into two subsequent parts, according to their label status. The labelled vectors will be used to predict the similarity score; while the unlabelled vectors will be delivered to the decoder for further feature training, which is introduced in the following section.

2) ENCODER-DECODER

a: ENCODER

The encoder module shares the same CNN model as described in Section IV-C.1. The main difference lies in the noising setting. In the encoder module, each CNN layer will be infected into a Gaussian random noise for further decoding. Therefore, the convolution in the encoder is represented as:

$$o = k^{\top} \cdot x + b + N(0, \sigma^2) \tag{6}$$

where $N(0, \sigma^2)$ refers to the Gaussian noise. The same noise is also added to the pooling layer and full connection layer. In addition, the maxpooling indices in the encoder, i.e., the locations of the maximum feature value in each pooling window, are preserved to facilitate the further decode processing.

b: DECODER

The decoder construction is inspired by the CNN-Encoder-Decoder [69]. As shown in Figure 5, it contains an upsampling layer and a special deconvolution layer. The upsampling layer decodes the maxpooling based on the max-pooling indices preserved in the encoder, and produces n

sparse feature maps. Afterwards, the deconvolution layer receives the maps and transforms them into dense ones relying on new convolution computation. The dense features are then averaged as a single output with the same feature size as the original clean input. Eventually, the encoder-decoder minimizes the difference between the clean input p_i and the reconstructed decoding output $\hat{p_i}$. The loss function for this part can be represented as the following equation:

$$\mathcal{L}_d = \frac{\lambda}{|P_u|} \sum_{i \in P_u} ||\hat{p_i} - p_i||_2^2 \tag{7}$$

where P_u refers to the set of unlabelled code pairs.

3) MODEL TRAINING

The training code clone datasets can be divided into two typical parts regrading whether they are labelled. The labelled part is captured from a commonly used code clone dataset - BigCloneBench [17], which is categorized into three types: T1-ST3 (NT1), MT3 and WT3/4 (NT2) and non-clone (NT3), depending on their clone similarity (see Section V-A). The unlabelled part is collected from Github and StackOverflow, which contains 20k pairs of code fragments (randomly captured from Ground Knowledge). Our goal is to clearly distinguish such three clone types via making full use of the labelled and unlabelled datasets, so as to provide fundamental computation evidence for further review sharing. The training processing can be concluded as the following two paths:

a: LABELLED PATH

Both of the clean and noising (infected in the training) labelled code pairs will pass through each layer of the CNN model, and eventually participate in the clone prediction.

b: UNLABELLED PATH

Unlabelled code pairs are corrupted by Gaussian noise when they pass through the CNN layers. Afterwards, they are fed into decoder module for denoising.

The CNN parts of labelled and unlabelled paths share the same parameters, so that the unlabelled path is able to contribute to the CNN feature learning. The entire loss function is the combination of the two loss functions from the ordinary CNN and the autoencoder model respectively.

$$\begin{aligned}\mathcal{L}_\theta &= \mathcal{L}_p + \lambda \mathcal{L}_d \\ &= \frac{1}{|P_l|} \sum_{i \in P_l} (Baseline_i - \tilde{y_i})^2 + \frac{\lambda}{|P_u|} \sum_{i \in P_u} ||\hat{p_i} - p_i||_2^2 \end{aligned} \tag{8}$$

In the training, the three clone types are assigned based on three baseline scores: 0.165, 0.495 and 0.825 respectively, to represent their similarity nature. For a targeted code pair, the final prediction score falls into three score regions: [0, 0.33), [0.33, 0.66), and [0.66, 1], to represent three clone types respectively. To overcome the over-fit risk, the $l2$ norm regularization (formally, $\frac{\lambda}{2m}||w||^2$) is added to the entire loss, to ensure that large weight parameters are penalized. Furthermore, we set the *dropout* rate [70] as 0.5 to further mitigate the over-fit and reduce the training cost.

D. REVIEW PROCESSING

We collected numerous of code review pairs as Ground Knowledge for further sharing (described in detail in Section V-A). However, original reviews in Ground Knowledge are miscellaneous, which should be re-organized for better presentation. Therefore, we conduct the following three types of review processing.

1) REVIEW CLASSIFICATION

Inspired by traditional code review classification works [8], [9], it is meaningful for the code projects to obtain informative knowledge from the shared reviews via fine-grained classification. The shared reviews are eventually classified as four representative types [9]: bug report, feature request, user experience and rating. We adopt typical classification algorithms to carry out the task automatically.

First, the labelled reviews in the [8] are treated as training dataset and represented as context-sensitive numerical vectors using Skip-gram. Afterwards, the vectors are fed into several common-used classifiers for multi-classification. The detailed comparative experiment process is introduced in Section V-D.

2) DEDUPLICATION

Deduplication for shared reviews is a necessary step, as the duplicated and poorly organized reviews seriously affect user comprehension. To measure duplication, We use the *Jaccard* distance in the implementation to compute the similarity between two review slices, which is defined as follows:

$$\mathcal{J} = \frac{A \cap B}{A \cup B} \tag{9}$$

where A and B are the sets of words from two different review slices. In practice, we first perform word segmentation for each review slice and then calculate the sentence similarity using the Jaccard distance.

3) THEME EXTRACTION

To provide more informative suggestions for shared reviews, the keywords reflecting the theme of targeted review slice are automatically extracted and tagged via LDA (Latent Dirichlet Allocation [71]) model. For each review slice, the LDA model returns a similarity probability for each theme and the one with top probability is chosen as its real theme.

The extracted themes and their proportions for review amount are listed as Table 1.

E. RANKING AND SEARCH

1) RANKING STRATEGY

For a given target code fragment, RSharer+ computes the difference between prediction score and baseline score of each clone fragment, ranks the results in ascendant order, and shares the reviews of the top n clone code fragments. In such

TABLE 1. Statistics of LDA themes.

Theme	Proportion(%)	Keywords
android	5.43	context, toast, bundle, support, oncreate
project management	17.99	dependency, artifactid, version, spring, maven
listener event	8.65	input, button, click, ajax, getelementbyid
background image	9.17	background, img, display, color, href
file	16.61	file, ioexception, close, line, map
vue	6.91	webpack, node_modules, export, module, vue
http request	5.41	response, request, http, post, json
react	3.57	props, react, document, getelementbyid, render
node	15.93	array, item, index, node, equals
JAVA GUI	10.34	awt, swing, avax, jpanel, frame

a way, RSharer+ provides users more choices which lead to better sharing result. By default, the ranking number n is empirically set to 3.

2) HEURISTIC SEARCH

In review sharing, it is necessary to search the clone code fragment from the Ground Knowledge. The code search is often implemented using iterative clone detection, which is time-consuming. Narrowing the search range to reduce the sharing cost is thus necessary and important. We achieve the narrowing goal relying on heuristic retrieve strategies. In more details, we adopted two types of heuristic filter factors proposed in [66], namely *code size* and *action token*. The filters divide the code space into groups that are labelled by the filter values, which is formally defined as follows:

$$\mathcal{G} = \mathcal{G}_{s_1} \cup \mathcal{G}_{s_2} \cup \cdots \cup \mathcal{G}_{s_n} \tag{10}$$

$$\mathcal{G} = \mathcal{G}_{a_1} \cup \mathcal{G}_{a_2} \cup \cdots \cup \mathcal{G}_{a_m} \tag{11}$$

where the equation (10) is the partition by code size and the equation (11) refers to the partition by action token, respectively. And the narrowed code set for searching becomes:

$$\mathcal{G}_{obj} = \mathcal{G}_{s_i} \cap \mathcal{G}_{a_j} \tag{12}$$

In practice, the code sizes are defined as 4 levels according to the line of code (LOC) (<20, 20-50, 50-100, 100-200 respectively). Note that the size range is loose, since a detailed size partition tends to lead into false negatives. In contrast, the action token strategy carries out the partition in a fine-grained way, where some typical features of the targeted code fragment (e.g., source and result) are token into consideration. In this process, we mainly manipulate the number and type of inputs/results for the partition. For instance, variable types with the same nature are categorized as the same group. If the code fragment does not consist of any source or result token, it is categorized as a unified independent group.

V. EXPERIMENTS

A. SUBJECTS

1) DATASET FOR CLONE DETECTION

The entire dataset can be divided into two parts according to the labelled status. The **labelled dataset** is constructed by the code pairs from popular clone dataset BigCloneBench [17], which covers 10 functionalities. The code pairs of the same functionality are stored in the same fold (in total 10 folds, indexed from #2 to #11, respectively). The labelled dataset covers 6+ million tagged true clone pairs and 260k tagged false clone pairs. The true clone pairs are labelled with 6 different tags: T1 (Type 1), T2 (Type 2), VST3 (Very Strong Type 3), ST3 (Strong Type 3), MT3 (Moderately Type 3), and WT3/4 (Weak Type 3 or Type 4), respectively. The similarity range for the code pairs with T1, T2, VST3 and ST3 tags is 70%-100%; for MT3 tagged pairs, the similarity range is 50%-70%; and for WT3/4, the range is 0-50%. Therefore, to facilitate the comparison, we categorize all the BigCloneBench data into 3 classes: the set of code pairs with tag T1-ST3 (class 1 clone, denoted as NT1), the set code pairs with tag MT3 and WT3/4 (class 2 clone, denoted as NT2), and the set of non-clone pairs (denoted as NT3). The **unlabelled dataset** is collected from code management platform Github and Stackoverflow, which consists of over 19k code fragments. We randomly select 20k code pairs across the code fragments to participate in the semi-supervised CNN model training.

The testing for comparative clone detection models contains two stages. The first stage aims to test the labelled data from BigCloneBench. To balance the training and testing number of NT1, NT2 and NT3, we randomly select 20k pairs as training data and 2k pairs as testing data from the three categories respectively. The second stage aims to test the data from Ground Knowledge (real world data). Due to the lack of labelled ground truth in Ground Knowledge, we manually checked and tagged 400 code pairs (from 10k unlaballed code pairs) as ground truth, where the NT1, NT2 and NT3 account for 100, 100 and 200, respectively. The numbers of code pairs in the training and test datasets are shown as column "Clone Detection" in Table 2.

In general, increasing training data will improve precision as well as mitigate over-fit risk. Therefore, to augment the size of the training data, in the review sharing, we use the entire listed data as the training dataset to learn the parameters of the final CNN model.

2) GROUND KNOWLEDGE

The Ground Knowledge consists of a large number of code review pairs (CRP), and serves as the domain dictionary to look up by the targeted code fragment. To establish

TABLE 2. Dataset Overview. #BCB and #PF refer to the code pairs collected from BigCloneBench and ground knowledge respectively.

Dataset	Clone Detection							Review Sharing	Review Classification
	#NT1		#NT2		#NT3		#Unlabelled		
	#BCB	#PF	#BCB	#PF	#BCB	#PF	#PF		
Training	20000	–	20000	–	20000	–	20000	–	4400 [8]
Testing	8400	100	45000	100	45000	200	–	200	200 (for 4 types)

applicable Ground Knowledge, we captured 27, 877 CRPs from two commonly used development platforms: Stackoverflow and Github. It is non-trivial task to capture the CRPs from large projects, since the reviews in these projects are often irrelevant to the detailed code. Although some recent works [6], [72] proposed methods for code localization, these works perform unstable in accuracy. Therefore, in practice, we only select those reviews associated with code fragments. Figure 6 is a typical example of the collected Ground Knowledge, where the texts at the top are brief description of the code fragments below.

Note that not all the captured CRPs can be directly used in the review sharing, as some code fragments are poorly presented, containing lexical errors or incomplete syntax expressions. Therefore, we need to first detect such incompleteness by lexical/syntax analyzer. We then repair the partial incomplete code fragments and discard others. At the end, there remain 19, 025 CRPs.

a: DATASET FOR REVIEW SHARING

As shown in column "Review Sharing" of Table 2, the testing dataset for review sharing contains 200 real world code fragments randomly collected from Ground Knowledge.

b: DATASET FOR REVIEW CLASSIFICATION

As shown in column "Review Classification" of Table 2, for review classification, all the classifiers are trained on the 4.4k labelled reviews described in [8]. In order to verify the effectiveness of RSharer+ in review classification, we also manually tag 50 reviews for each review type as ground truth, i.e., totally 200 reviews for 4 types: bug report, feature request, user experience and rating.

B. METRICS AND BASELINE

1) METRICS

Both of the clone detection and review classification modules are evaluated using 3 typical evaluation metrics: precision, recall and f1-score. Since the clone detection is actually a multi-classification problem, we re-define the three metrics with respect to the classification reality [73]. Let T be the number of classes, the matrix $V = T \times T$ is the number matrix of classification results. The i-th row of V represents the true class, and correspondingly the j-th column of V represents the estimated class. Thus an element v_{ij} is the number of classification that true class i is estimated as class j. On top of that, the *precision for class j* ($Precision_j$) and the *total precision* ($Precision$) can be defined.

The $precision_j$ indicates the percentage of clone pairs that are correctly classified into class j among all the clone pairs classified as class j.

$$Precision_j = \frac{v_{jj}}{\sum_{i=1}^{T} v_{ij}} \tag{13}$$

The *total precision* is the precision for all the classes, computed as the mean of all the precision values.

Note that not all the pairs of two code fragments in BigCloneBench are explicitly tagged. Hence, the partially classified pairs cannot be categorized into their real classes, which prevents the traditional methods from computing the precision [34], [66]. However, such problem is not an issue against the equation (13), since the objective pairs handled in the equation (13) are clearly tagged (see the definition of v_{ij}) and the pairs without any tag are directly discarded.

The *recall for class i* ($Recall_i$) and the total recall are defined in the following formula.

$$Recall_i = \frac{v_{ii}}{\sum_{j=1}^{T} v_{ij}} \tag{14}$$

The $Recall_i$ indicates the percentage of clone pairs that are correctly classified into class i among all the pairs of class i.

Similar to the *total precision*, the *total recall* is the recall for all the classes, computed as the mean of all the recalls.

Finally, *f1-score for class i* ($F1 - score_i$) is defined as the harmonic mean of the precision and recall, which represents the trade-off between precision and recall.

$$F1 - score_i = \frac{2 \times Precision_i \times Recall_i}{Precision_i + Recall_i} \tag{15}$$

The *total F1-score* is calculated as the mean of all the F1-scores.

2) BASELINE

The baselines for different evaluation stages are introduced respectively in this section.

a: CLONE DETECTION

We investigated the state-of-the-art code clone detection works, including SourcererCC [29], NiCad [28], Deckard [33], CClearner [34], and Oreo [66]. In particular, SourcererCC, NiCad and Deckard are implemented using the traditional tree-based and graph-based techniques which are difficult be adopted in our multi-classification task; while CClearner and Oreo are implemented using deep learning techniques which can be extended to the multi-classification task in our work. Hence, the CClearner and Oreo are selected as two baselines in clone detection. Apart from that, we also

If you already have the content you want to write to the file (and not generated on the fly), the `java.nio.file.Files` addition in Java 7 as part of native I/O provides the simplest and most efficient way to achieve your goals.

Basically creating and writing to a file is one line only, moreover **one simple method call!**

The following example creates and writes to 6 different files to showcase how it can be used:

```
Charset utf8 = StandardCharsets.UTF_8;
List<String> lines = Arrays.asList("1st line", "2nd line");
byte[] data = {1, 2, 3, 4, 5};

try {
    Files.write(Paths.get("file1.bin"), data);
    Files.write(Paths.get("file2.bin"), data,
            StandardOpenOption.CREATE, StandardOpenOption.APPEND);
    Files.write(Paths.get("file3.txt"), "content".getBytes());
    Files.write(Paths.get("file4.txt"), "content".getBytes(utf8));
    Files.write(Paths.get("file5.txt"), lines, utf8);
    Files.write(Paths.get("file6.txt"), lines, utf8,
            StandardOpenOption.CREATE, StandardOpenOption.APPEND);
} catch (IOException e) {
    e.printStackTrace();
}
```

FIGURE 6. An example of collected ground knowledge.

implement two Word2Vec based methods: Word2Vec+SVM and Word2Vec+CNN as baselines to compare with the proposed semi-supervised CNN.

b: REVIEW RANKING

In order to validate the rationality of review ranking, we have implemented comparative review sharing systems that adopt CCLearner (CCSharer), Oreo (OreoSharer), WV+CNN (RSharer) and Semi-CNN (RSharer+) as the clone detection kernel, respectively.

c: REVIEW CLASSIFICATION

In review classification, to inspect a superior classification method, we implement four Word2Vec based methods and test them on the manually tagged ground truth (200 labelled review slices described in Section V-A.1).

C. EXPERIMENTAL SETTING

All the experiments are conducted using 64-bit Windows 10OS, IntelXeon E5-2678 CPU and 64G memory. In addition, the neural networks are implemented using Tensorflow 1.12 with aNvidia TITAN Xp GPU. We use ASTView, an eclipse built-in plugin, to extract the ASTs for both the Ground Knowledge and the target code fragment. To optimize the Word2Vec training, the *negative sampling* [64] is used. We set the size of negative samples to 10, the window size to 5, the embedding dimension to 64, and the minimum word frequency (*min_count*) to 3. In the CNN models (including the CNN module of semi-supervised CNN and Word2Vec+CNN), we set the batch size to 32, the epoch number to 200, the learning ratio to $1e^{-3}$, the dropout ratio to 0.5, the filter number for each window size to 200 and the node number of the full connection to 640. The stride sizes in both convolution layer and pooling layer are set to the default value 1. In addition, we keep the input sequence length (each code sentence length) as 400 because of the requirement of fixed size for CNN input. Moreover, since the input vectors produced by the Word2Vec may contain negative values and the zero padding strategy may incur negative impacts in the pooling layer, we use the "VALID" (no padding) mode in the phases of convolution and pooling, where the extra windows are directly discarded.

The original comparison methods (CCLearner and Oreo) only target at binary classification for clone and non-clone, and thus are difficult to be directly applied to the multi-classification problem. Therefore, we modify the classifiers of the two methods to suit for our comparison. For CCLearner, we replace the original two types of labels (clone and non-clone) with the new three types (NT1, NT2 and NT3) in the training phase. For Oreo, we consider the matched clones by the metric hash filter as the NT1 clones; non-matched clones, but being detected by DNN models, as the NT2 clones; the remaining ones as the NT3 clones. Note that we do not change any other setting described in their corresponding published papers [34], [66] when conducting the experiments.

FIGURE 7. Comparison results of clone detection for BigCloneBench. WV+SVM denotes SVM with Word2Vec embedding; WV+CNN denotes CNN with Word2Vec embedding; Semi+CNN denotes semi-supervised CNN with Word2Vec embedding.

FIGURE 8. Comparison results of clone detection for ground knowledge. WV+SVM, WV+CNN and Semi+CNN denote the same methods as Figure 7.

For review classification, the review texts are pre-processed via discarding the stop words and then embedded by the Skip-gram. Thus, each word is transformed into a context sensitive 200 dimensional vector. Afterwards, the Bayes and SVM classifiers directly work on the embedded vectors. For CNN, the convolution is computed on the $200 \times k$ ''images'', where the k is a fixed number according to the sentence segmentation. In practice, the k is set as 50, and other settings are the same as ordinary CNN in clone detection described in Section IV-C.1.

D. EVALUATION

To systematically evaluate the RSharer+, we aim to answer the following four research questions from the perspectives of clone detection, review ranking, sharing presentation and time costs, respectively.

RQ1: How does the autoencoder-based semi-supervised CNN perform in detecting clone pairs compared with the state-of-the-art approaches?

We progressively test the clone detection capability at two different stages: 1) for existing dataset (BigCloneBench); 2) for real collected dataset (Ground Knowledge), and correspondingly divide the RQ1 into two sub-questions.

RQ1-1: Clone Detection for BigCloneBench.

The overall clone detection result for 98.4k (see Table 2) labelled clone pairs in BigCloneBench is shown in Figure 7. We can see that the CNN based approaches (including WV+CNN and Semi-supervised CNN) outperform other methods, especially in detecting NT2 and NT3 types. The CNN-based approaches extract latent features that would impact prediction results using deep learning structure; while in other two methods, the features are handcrafted, which can hardly cover the comprehensive effective factors. It also can be seen that the semantic information at both word level and sentence level (both are considered in Word2Vec based approach) is especially useful in detecting semantic-based clone types. For NT1 clones, the advantages of our approach is minor since for all methods, it is straightforward to detect such clones. Regarding the comparison between WV+CNN and Semi-supervised CNN, the performance gap is trivial, which illustrates that the semi-supervision training for unlabelled data makes little difference when testing the ground truth from labelled dataset.

RQ1-2: Clone Detection for Ground Knowledge.

Figure 8 shows the clone detection results for tagged 400 code pairs from Ground Knowledge. From the results,

TABLE 3. Result details for clone detection in BigCloneBench. 1/2/3-E is the number of the estimated clone pairs for NT1,NT2 and NT3 respectively; 1/2/3-T refers to the number of true clone pairs for NT1,NT2 and NT3 respectively; WV+SVM, WV+CNN and Semi+CNN denote the same methods as in Figure 7. The bold numbers refer to the number of pairs that are correctly classified.

(k)	CCLearner			Oreo			WV+SVM			WV+CNN			Semi-CNN			Method
	1-E	2-E	3-E	1-E	2-E	3-E	1-E	2-E	3-E	1-E	2-E	3-E	1-E	2-E	3-E	Overall
1-T	**8**	0.3	0.05	**8.1**	0.25	0.05	**8.26**	0.09	0.05	**8.34**	0.02	0.04	**8.32**	0.03	0.05	8.40
2-T	0.62	**22.6**	21.7	0.31	**26.6**	18.1	0.14	**31.0**	13.8	0.07	**32.4**	12.5	0.11	**35.0**	9.91	45
3-T	0.07	18.6	**26.3**	0.04	16.6	**28.3**	0.04	14.6	**30.3**	0.04	8.1	**36.9**	0.06	8.38	**36.6**	45
Overall	8.74	41.6	48	8.5	43.5	46.5	8.4	45.8	44.2	8.5	40.6	49.4	8.49	43.4	46.5	98.4

TABLE 4. Result Details for Clone Detection in Ground Knowledge. 1/2/3-E is the number of the estimated clone pairs for NT1, NT2 and NT3 respectively; 1/2/3-T refers to the number of true clone pairs for NT1, NT2 and NT3 respectively; WV+SVM, WV+CNN and Semi+CNN denote the same methods as in Figure 7. The bold numbers refer to the number of pairs that are correctly classified.

	CCLearner			Oreo			WV+SVM			WV+CNN			Semi-CNN			Method
	1-E	2-E	3-E	1-E	2-E	3-E	1-E	2-E	3-E	1-E	2-E	3-E	1-E	2-E	3-E	Overall
1-T	**89**	2	9	**84**	3	13	**82**	3	15	**86**	6	8	**92**	3	5	100
2-T	8	**44**	48	14	**48**	38	14	**62**	24	8	**60**	32	8	**74**	18	100
3-T	4	68	**128**	2	56	**142**	4	53	**143**	6	23	**151**	2	22	**176**	200
Overall	101	114	185	100	107	193	100	118	182	100	109	191	102	109	209	400

semi-supervised CNN evidently outperforms all of comparative methods including WV+CNN in terms of the precision, recall and f1-score. The reason lies in two aspects. On the one hand, semi-supervision model makes the training data tend to have the same manifold distribution as testing data. In contrast, the comparative methods conduct the training on the data purely obtained from BigCloneBench, which is naturally differently distributed with testing dataset in the manifold space. On the other hand, the semi-supervised model exploits an encoder-decoder structure to learn the essential features of unlabelled distribution, which improves contribution of unlabelled data.

RQ2: How does RSharer+ perform for review ranking?

To answer RQ2, we have conducted comparative sharing experiments on random 200 real-world code fragments. Figure 10 shows the sharing results which have been confirmed manually. Out of the 200 targeted code fragments, 68 obtain reasonable shared reviews by CCSharer (based on CCLearner); 67 achieve reasonable results by OreoSharer (based on Oreo); 97 obtain reasonable reviews by RSharer (based on WV+CNN); 113 obtain reasonable reviews by RSharer+ (based on Semi-CNN).

A sharing case is identified as a reasonable one if at least one out of the three ranking results is reasonable. A reasonable result means that the shared review is exactly for the code fragment or at least closely relevant to the code fragment.

It can be seen that RSharer+ is able to detect more true clone cases and more reasonable sharing cases than the comparative methods, but gets lower R-Rate than CCSharer and OreoSharer. Since the four systems use exactly the same sharing ranking strategy, the difference of sharing results is attributed to the clone detection. It suggests that an accurate clone search will significantly benefit the final sharing result. Regarding the low R-Rate result, the reason lies in that most true clones detected by CCSharer and OreoSharer are of type NT1; while most of the clones detected by RSharer and RSharer+ are of type NT1 or NT2. It is natural that NT1 clones share more reasonable reviews than NT2. Despite the shortage in R-Rate, RSharer+ still generates more effective review sharing cases, thanks to its fine performance on NT2 clone search.

Figure 9 demonstrates a typical review sharing case in which RSharer+ successfully shares a reasonable review but all the other three methods fail. The targeted code fragment aims to initialize a google map. The review shared by RSharer+ (in the blue frame) describes a boxing problem when setting directions on the map, which is closely related to the targeted code. Nevertheless, the reviews shared by other sharers present totally different topics with the targeted code fragment (in the red frames). It indicates that the targeted code fragment is more fitting for the data distribution of the collected unlabelled dataset, which is successfully extracted only by the RSharer+.

The Figure 9 also presents some detailed information (shown in the yellow frame), such as ranking list, code similarity, review type and review theme, whose effectiveness is analyzed in the following sections.

RQ3: How are the shared reviews presented to facilitate review analysis?

In order to better present the shared reviews, two types of further analysis are conducted for the shared reviews: review classification and theme extraction.

Review Classification.

Figure 11 and Table 5 list the comparison classification results and the details. It can be observed that the WV+CNN* evidently outperforms other methods in classification of bug report, user experience and rating. Nevertheless, for feature request, WV+SVM* is significantly superior than the WV+CNN* in terms of the three metrics. The reason might lie in the feature request reviews lack evident local features as the other three types, which weakens the effect of

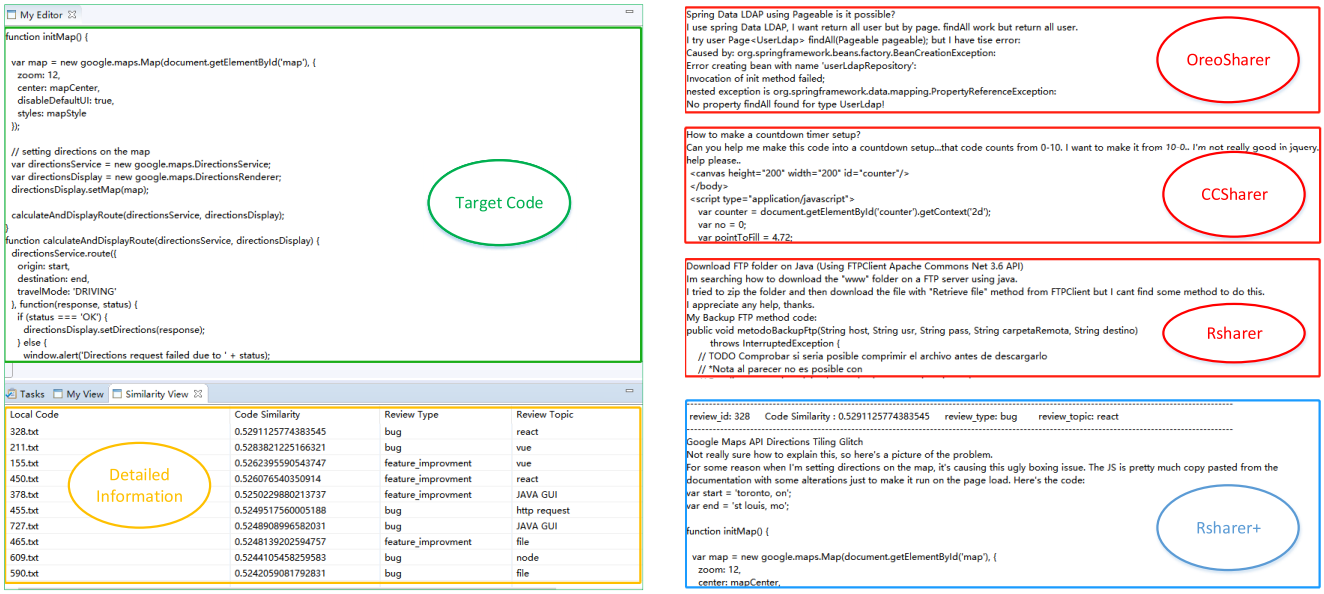

FIGURE 9. Sharing examples(Correct example).

TABLE 5. Result details for review classification. B/F/U/R-E is the number of the estimated clone pairs for B, F, U, R; B/F/U/R-T denotes the number of the true clone pairs for B, F, U, R; WV+Bayes, WV+SVM* and WV+CNN* are the Bayes classifier with Word2Vec embedding, SVM classifier with Word2Vec embedding and CNN classifier with Word2Vec embedding, respectively. The bold numbers refer to the number of correctly classified pairs.

	WV+Bayes				WV+SVM*				WV+CNN*				Method
	B-E	F-E	U-E	R-E	B-E	F-E	U-E	R-E	B-E	F-E	U-E	R-E	Overall
B-T	**40**	7	2	1	**33**	7	2	8	**42**	7	0	1	50
F-T	4	**35**	1	10	5	**42**	1	2	7	**23**	10	10	50
U-T	3	19	**20**	8	3	2	**32**	13	3	4	**43**	0	50
R-T	1	7	0	**42**	2	1	5	**42**	0	0	2	**48**	50
Overall	48	68	23	61	43	52	40	65	52	34	55	59	200

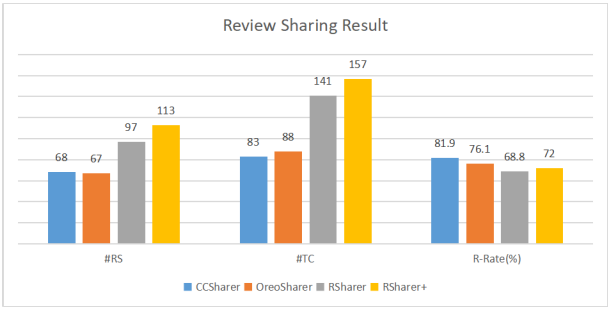

FIGURE 10. Review sharing Result. #RS is the number of the reasonable sharing cases; #TC is the number of the detected true clone cases; and R-Rate is the proportion of #RS among #TC.

CNN. Therefore, we employ both of the WV+CNN* and WV+SVM* in practice to make full use of their advantages, where we use WV+SVM* to obtain the feature request result and WV+CNN* for other three types of reviews. In this process, there are 134 conflicts between the two methods, which are resolved by hand.

Theme Extraction: To provide informatively presentation, the LDA model is employed to conduct theme extraction for each review slice. A typical example of the theme extraction is shown as Figure 9, where the review is tagged with "react" theme (the first line in the yellow frame) that corresponds to the key word "document", and the review goal is indeed to find a reaction resolution about the documentation of map API, so the theme extraction provides a reasonable suggestion for users and developers. In a global scope, by our investigation, the error rate for the theme extraction is acceptable (33/287), and the observed error themes have been corrected by hand.

RQ4: How is the time cost of compared methods in clone detection and review sharing?

Since the AST token extraction, clone model training, review classification and theme extraction can be carried out offline, the time costs of these stages impact trivially for users. Therefore, we only concentrate on the time performance of the real clone detection and ranking. Note that since the code fragments and its reviews are captured one by one in the Ground knowledge, the clone detection and review ranking can be treated as at the same stage. We record the time costs of the 200 code review sharing in RQ2 for each of the comparative method, and compute the average time cost for each review sharing which are listed in Figure 12. We use four computers with the same environment to complete the

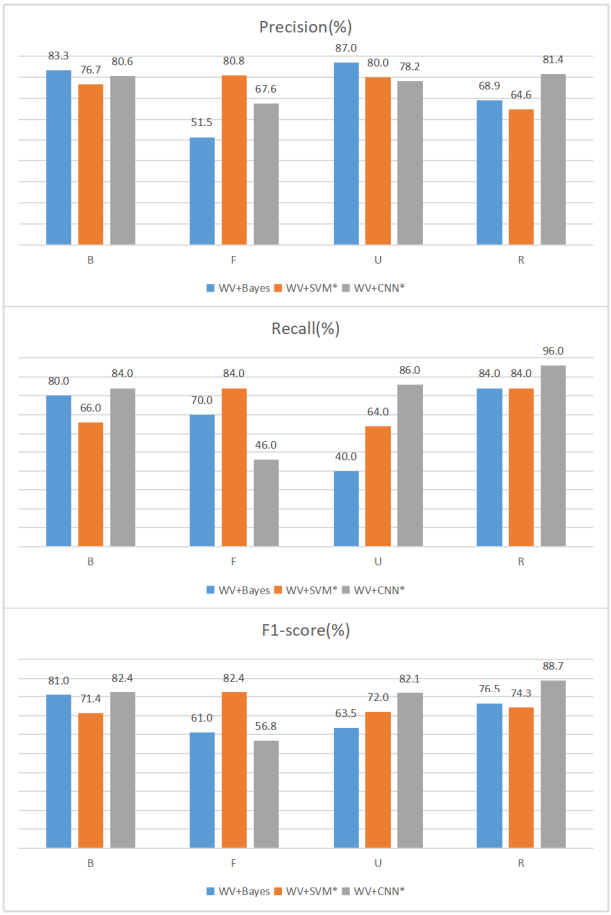

FIGURE 11. Comparison results of review classification. B, F, U, R refer to bug report, feature request, user experience, rating, respectively.

testing in a parallel way. We can observe that OreoSharer achieves the lowest time cost, owing to its effective filtering strategy. It can be seen that the RSharer and RSharer+ also benefit a lot from the heuristic filtering strategy, resulting in relatively lower costs than CCSharer. It is worth noticing that the time costs of all the four systems are challenging for real online sharing task, and the time cost would increase along with the scale of Ground Knowledge. Hence, it is required to explore more performance optimization strategies to concretely reduce the cost, which is discussed in the Section V-E.

E. THREATS TO VALIDITY

In this section, we discuss several threats that potentially impact the validity of our work, which will guide the future work.

1) TIME COST

Due to a large amount of CRPs in Ground Knowledge, code search is time consuming though some heuristic search strategies are adopted. The average time cost for a sharing request reaches 28.3 min, which blocks our approach to be applied in real-world scenario. The performance consideration also limits the extension of Ground Knowledge. In the future, we plan to optimize the time performance via improving both hardware devices and ranking algorithm.

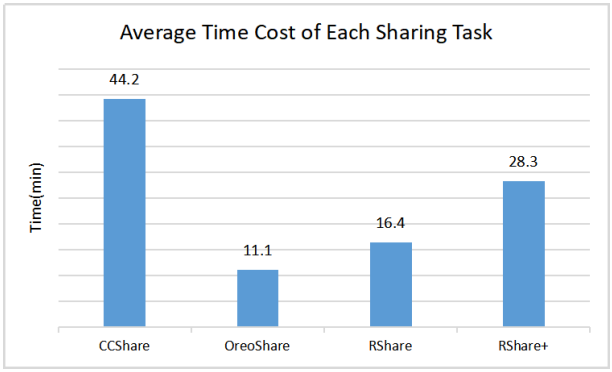

FIGURE 12. Time cost.

F. EVALUATION

The evaluation of the sharing ranking results is based on manual analysis, which is inevitably influenced by the analyser's domain knowledge. We reduce such influence by synthesizing the opinions of all the evaluation participants (including 3 laboratory assistants and 2 senior developers). In addition, we do not adopt 10-fold cross validation in the clone detection and review classification, since the test data is labelled by hand, which is not enough for training.

G. USABILITY

Currently, RSharer+ only accepts Java inputs, which limits its language applicability. Moreover, current RSharer+ is only available for the code fragments, and it is still challenging to share for complete software projects due to the dramatic increase of resource costs. We plan to explore effective pruning strategies to fit the project-level sharing in the future.

VI. CONCLUSION

In this work, we present RSharer+, a new semi-supervised CNN based review sharing technique. Through the technique, projects without reviews are able to obtain sufficient reviews not only fitting for code semantics but also beneficial to further improvement.

RSharer+ adopts code clone detection to search suitable reviews from existing code projects. To make full use of unlabelled code projects collected from real code platforms, RSharer+ implements a semi-supervised CNN encoder-decoder, in which both of the labelled and unlabelled data contribute to the feature learning, and eventually enhance the accuracy of clone detection. Apart from that, in order to better present the sharing results, theme-based ranking strategy is implemented using LDA model and informative classification. Again, the heuristic sifting strategy is employed to reduce the time cost of clone search.

Extensive experiments are conducted to verify the effectiveness of RSharer+. First, in the code clone detection on BigCloneBench, experimental results demonstrate that

the CNN based clone computation outperforms traditional methods. Then, in the clone detection on collected code repositories, the autoencoder-based semi-supervised algorithm validates its effectiveness when comparing with original CNN clone detection. Afterwards, through sharing for hundreds of real projects, RSharer+ achieves more reasonable sharing reviews by manual confirmation. Finally, the theme-based ranking strategy is also verified to be effective for better review presentation via comparative experiments.

REFERENCES

[1] Y. Yu, H. Wang, G. Yin, and T. Wang, "Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment?" *Inf. Softw. Technol.*, vol. 74, pp. 204–218, Jun. 2016.

[2] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K.-I. Matsumoto, "Who should review my code? A file location-based code-reviewer recommendation approach for Modern Code Review," in *Proc. IEEE 22nd Int. Conf. Softw. Anal., Evol., Reeng. (SANER)*, Mar. 2015, pp. 141–150.

[3] A. Di Sorbo, S. Panichella, C. V. Alexandru, J. Shimagaki, C. A. Visaggio, G. Canfora, and H. C. Gall, "What would users change in my app? Summarizing app reviews for recommending software changes," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, 2016, pp. 499–510.

[4] O. Kononenko, O. Baysal, and M. W. Godfrey, "Code review quality: How developers see it," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng. (ICSE)*, May 2016, pp. 1028–1038.

[5] N. Chen, J. Lin, S. C. H. Hoi, X. Xiao, and B. Zhang, "AR-miner: Mining informative reviews for developers from mobile app marketplace," in *Proc. 36th Int. Conf. Softw. Eng. (ICSE)*, 2014, pp. 767–778.

[6] F. Palomba, P. Salza, A. Ciurumelea, S. Panichella, H. Gall, F. Ferrucci, and A. De Lucia, "Recommending and localizing change requests for mobile apps based on user reviews," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. (ICSE)*, May 2017, pp. 106–117.

[7] Y. Zhou, Y. Su, T. Chen, Z. Huang, H. Gall, and S. Panichella, "User review-based change file localization for mobile applications," 2019, *arXiv:1903.00894*. [Online]. Available: https://arxiv.org/abs/1903.00894

[8] W. Maalej and H. Nabil, "Bug report, feature request, or simply praise? On automatically classifying app reviews," in *Proc. IEEE 23rd Int. Requirements Eng. Conf. (RE)*, Aug. 2015, pp. 116–125.

[9] S. Panichella, A. Di Sorbo, E. Guzman, C. A. Visaggio, G. Canfora, and H. C. Gall, "How can i improve my app? Classifying user reviews for software maintenance and evolution," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2015, pp. 281–290.

[10] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proc. 26th Conf. Program Comprehension (ICPC)*, 2018, pp. 200–210.

[11] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation with hybrid lexical and syntactical information," *Empirical Softw. Eng.*, to be published.

[12] T. Haije, B. O. K. Intelligentie, E. Gavves, and H. Heuer, "Automatic comment generation using a neural translation model," *Inf. Softw. Technol.*, vol. 55, no. 3, pp. 258–268, 2016.

[13] W. Zheng, H.-Y. Zhou, M. Li, and J. Wu, "Code attention: Translating code to comments by exploiting domain features," 2017, *arXiv:1709.07642*. [Online]. Available: https://arxiv.org/abs/1709.07642

[14] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proc. 54th Annu. Meeting Assoc. for Comput. Linguistics*, vol. 1, 2016, pp. 2073–2083.

[15] E. Wong, T. Liu, and L. Tan, "CloCom: Mining existing source code for automatic comment generation," in *Proc. IEEE 22nd Int. Conf. Softw. Anal., Evol., Reeng. (SANER)*, Mar. 2015, pp. 380–389.

[16] C. Guo, D. Huang, N. Dong, Q. Ye, J. Xu, Y. Fan, H. Yang, and Y. Xu, "Deep review sharing," in *Proc. IEEE 26th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Feb. 2019, pp. 61–72.

[17] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, Sep. 2014, pp. 476–480.

[18] J. Svajlenko and C. K. Roy, "Evaluating clone detection tools with BigCloneBench," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2015, pp. 131–140.

[19] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, 2016, pp. 87–98.

[20] J. H. Johnson, "Identifying redundancy in source code using fingerprints," in *Proc. 1993 Conf. Centre for Adv. Stud. Collaborative Res., Softw. Eng.*, Vol. 1, 1993, pp. 171–183.

[21] J. H. Johnson, "Visualizing textual redundancy in legacy source," in *Proc. Conf. Centre Adv. Stud. Collaborative Res.*, 1994, p. 32.

[22] B. S. Baker, "A program for identifying duplicated code," *Comput. Sci. Statist.*, vol. 24, p. 49, Mar. 1993.

[23] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Proc. IEEE Int. Conf. Softw. Maintenance (ICSM). Softw. Maintenance Bus. Change*, Aug./Sep. 1999, pp. 109–118.

[24] J. H. Johnson, "Substring matching for clone detection and change tracking," in *Proc. ICSM*, vol. 94, 1994, pp. 120–126.

[25] B. Baker, "On finding duplication and near-duplication in large software systems," in *Proc. 2nd Work. Conf. Reverse Eng.*, Nov. 2002, pp. 86–95.

[26] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding copy-paste and related bugs in large-scale software code," *IIEEE Trans. Softw. Eng.*, vol. 32, no. 3, pp. 176–192, Mar. 2006.

[27] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code," *IIEEE Trans. Softw. Eng.*, vol. 28, no. 7, pp. 654–670, Jul. 2002.

[28] C. Roy and J. Cordy, "NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *Proc. 16th IEEE Int. Conf. Program Comprehension*, Jun. 2008, pp. 172–181.

[29] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "SourcererCC: Scaling code clone detection to big-code," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng. (ICSE)*, May 2016, pp. 1157–1168.

[30] P. Wang, J. Svajlenko, Y. Wu, Y. Xu, and C. K. Roy, "Ccaligner: A token based large-gap clone detector," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 1066–1077.

[31] W. Yang, "Identifying syntactic differences between two programs," *Softw., Pract. Exper.*, vol. 21, no. 7, pp. 739–755, Jul. 1991.

[32] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proc. Int. Conf. Softw. Maintenance*, Nov. 2002, pp. 368–377.

[33] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," in *Proc. 29th Int. Conf. Softw. Eng. (ICSE)*, May 2007, pp. 96–105.

[34] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, "CcLearner: A deep learning-based clone detection approach," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2017, pp. 249–260.

[35] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "Deep learning similarities from different representations of source code," in *Proc. 15th Int. Conf. Mining Softw. Repositories (MSR)*, 2018, pp. 542–553.

[36] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *Proc. ACM/IEEE 30th Int. Conf. Softw. Eng. (ICSE)*, May 2008, pp. 321–330.

[37] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *Proc. Int. Static Anal. Symp.* Berlin, Germany: Springer, 2001, pp. 40–56.

[38] J. Krinke, "Identifying similar code with program dependence graphs," in *Proc. 8th Work. Conf. Reverse Eng.*, Nov. 2002, pp. 301–309.

[39] C. Liu, C. Chen, J. Han, and P. S. Yu, "GPLAG: Detection of software plagiarism by program dependence graph analysis," in *Proc. 12th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2006, pp. 872–881.

[40] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on Android markets," in *Proc. 36th Int. Conf. Softw. Eng. (ICSE)*, 2014, pp. 175–186.

[41] N. H. Pham, H. A. Nguyen, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen, "Complete and accurate clone detection in graph-based models," in *Proc. IEEE 31st Int. Conf. Softw. Eng.*, May 2009, pp. 276–286.

[42] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.TOPLAS*, vol. 9, no. 3, pp. 319–349, Jul. 1987.

[43] F. E. Allen, "Control flow analysis," *ACM Sigplan Notices*, vol. 5, no. 7, pp. 1–19, 1970.

[44] A. D. Sorbo, S. Panichella, C. A. Visaggio, M. D. Penta, G. Canfora, and H. C. Gall, "Development emails content analyzer: Intention mining in developer discussions (T)," in *Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2015, pp. 12–23.
[45] A. Bacchelli, M. Lanza, and R. Robbes, "Linking e-mails and source code artifacts," in *Proc. ACM/IEEE 32nd Int. Conf. Softw. Eng.*, vol. 1, May 2010, pp. 375–384.
[46] A. Di Sorbo, S. Panichella, C. A. Visaggio, M. Di Penta, G. Canfora, and H. Gall, "DECA: Development Emails content analyzer," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng. Companion (ICSE-C)*, May 2016, pp. 641–644.
[47] C. Vassallo, S. Panichella, M. Di Penta, and G. Canfora, "CODES: Mining source code descriptions from developers discussions," in *Proc. 22nd Int. Conf. Program Comprehension (ICPC)*, 2014, pp. 106–109.
[48] S. Panichella, J. Aponte, M. Di Penta, A. Marcus, and G. Canfora, "Mining source code descriptions from developer communications," in *Proc. 20th IEEE Int. Conf. Program Comprehension (ICPC)*, Jun. 2012, pp. 63–72.
[49] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *Proc. 28th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2013, pp. 345–355.
[50] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshynanyk, and A. De Lucia, "How to effectively use topic models for software engineering tasks? An approach based on Genetic Algorithms," in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, May 2013, pp. 522–531.
[51] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*, vol. 463. New York, NY, USA: ACM Press, 1999.
[52] M. White, C. Vendome, M. Linares-Vasquez, and D. Poshyvanyk, "Toward deep learning software repositories," in *Proc. IEEE/ACM 12th Work. Conf. Mining Softw. Repositories*, May 2015, pp. 334–345.
[53] T. Beltramelli, "pix2code: Generating code from a graphical user interface screenshot," in *Proc. ACM SIGCHI Symp. Eng. Interact. Comput. Syst.*, 2018, p. 3.
[54] L. Mou, R. Men, G. Li, L. Zhang, and Z. Jin, "On end-to-end program generation from user intention by deep neural networks," 2015, *arXiv:1510.07211*. [Online]. Available: https://arxiv.org/abs/1510.07211
[55] C. Chen, T. Su, G. Meng, Z. Xing, and Y. Liu, "From UI design image to gui skeleton: A neural machine translator to bootstrap mobile gui implementation," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 665–676.
[56] H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source Code," in *Proc. 26th Int. Joint Conf. Artif. Intell.*, Aug. 2017, pp. 3034–3040.
[57] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *Proc. 40th Int. Conf. Softw. Eng. (ICSE)*, 2018, pp. 933–944.
[58] X. Gu, H. Zhang, D. Zhang, and S. Kim, "DeepAM: Migrate APIs with multi-modal sequence to sequence learning," 2017, *arXiv:1704.07734*. [Online]. Available: https://arxiv.org/abs/1704.07734
[59] T. D. Nguyen, A. T. Nguyen, H. D. Phan, and T. N. Nguyen, "Exploring API embedding for API usages and applications," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. (ICSE)*, May 2017, pp. 438–449.
[60] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *Proc. Int. Conf. Mach. Learn.*, 2016, pp. 2091–2100.
[61] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proc. 13th AAAI Conf. Artif. Intell.*, 2016.
[62] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Combining deep learning with information retrieval to localize buggy files for bug reports (N)," in *Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2015, pp. 476–481.
[63] H. Peng, L. Mou, G. Li, Y. Liu, L. Zhang, and Z. Jin, "Building program vector representations for deep learning," in *Int. Conf. Knowl. Sci., Eng. Manage.* Berlin, Germany: Springer, 2015, pp. 547–553.
[64] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 2013, pp. 3111–3119.
[65] Y. Goldberg and O. Levy, "word2vec explained: Deriving Mikolov's negative-sampling word-embedding method," 2014, *arXiv:1402.3722*. [Online]. Available: https://arxiv.org/abs/1402.3722
[66] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. V. Lopes, "Oreo: Detection of clones in the twilight zone," in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2018, pp. 354–365.
[67] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P.-A. Manzagol, "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion," *J. Mach. Learn. Res.*, vol. 11, no. 12, pp. 3371–3408, Dec. 2010.
[68] Y. Kim, "Convolutional neural networks for sentence classification," 2014, *arXiv:1408.5882*. [Online]. Available: https://arxiv.org/abs/1408.5882
[69] V. Badrinarayanan, A. Kendall, and R. Cipolla, "SegNet: A deep convolutional encoder-decoder architecture for image segmentation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 39, no. 12, pp. 2481–2495, Dec. 2017.
[70] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," 2012, *arXiv:1207.0580*. [Online]. Available: https://arxiv.org/abs/1207.0580
[71] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent Dirichlet allocation," *J. Mach. Learn. Res.*, vol. 3, pp. 993–1022, Mar. 2003.
[72] M. Paixao, J. Krinke, D. Han, and M. Harman, "Crop: Linking code reviews to source code changes," in *Proc. 15th Int. Conf. Mining Softw. Repositories*, 2018, pp. 46–49.
[73] B. Xu, D. Ye, Z. Xing, X. Xia, G. Chen, and S. Li, "Predicting semantically linkable knowledge in developer online forums via convolutional neural network," in *Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Aug. 2016, pp. 51–62.

CHENKAI GUO (Member, IEEE) received the Ph.D. degree from Nankai University, in 2017. He is an Assistant Professor with the College of Computer Science, Nankai University. His research interests include software analysis on mobile apps, information security, and intelligent software engineering.

HUI YANG received the B.E. degree from the Nanjing University of Aeronautics and Astronautics, in 2018. She is currently pursuing the master's degree with the College of Computer Science, Nankai University. Her research interests include mobile apps analysis and intelligent software engineering.

DENGRONG HUANG (Member, IEEE) received the B.E. degree from Nankai University, in 2017, where she is currently pursuing the master's degree with the College of Computer Science. Her research interests include mobile apps analysis and intelligent software engineering.

JIANWEN ZHANG received the B.E. degree from the Hebei University of Technology, in 2019. She is currently pursuing the master's degree with the College of Artificial Intelligence, Nankai University. Her research interests include mobile apps analysis and intelligent software engineering.

NAIPENG DONG received the bachelor's degree from Shandong University, China, and the master's and Ph.D. degrees from the University of Luxembourg. She is a Research Fellow of the School of Computing, National University of Singapore. Her research interests include automatic formal verification of security and privacy in cryptographic protocols, Android applications, and blockchain systems.

JING XU (Member, IEEE) received the Ph.D. degree from Nankai University, in 2003. She is currently a Professor with the College of Artificial Intelligence, Nankai University. Her current research interests include intelligent software engineering and medical data analysis. She received the Second Prize of the Tianjin Science and Technology Progress Award, in 2017 and 2018.

JINGWEN ZHU received the master's degree from Nankai University, in 2018. She is an Assistant Experimentalist with the College of Software, Nankai University. Her research interests include software analysis, information security, and intelligent software engineering.