Towards One Reusable Model for Various Software Defect Mining Tasks

Heng-Yi Li, Ming Li(✉), and Zhi-Hua Zhou

National Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China
{lihy,lim,zhouzh}@lamda.nju.edu.cn

Abstract. Software defect mining is playing an important role in software quality assurance. Many deep neural network based models have been proposed for software defect mining tasks, and have pushed forward the state-of-the-art mining performance. These deep models usually require a huge amount of task-specific source code for training to capture the code functionality to mine the defects. But such requirement is often hard to be satisfied in practice. On the other hand, lots of free source code and corresponding textual explanations are publicly available in the open source software repositories, which is potentially useful in modeling code functionality. However, no previous studies ever leverage these resources to help defect mining tasks. In this paper, we propose a novel framework to learn one reusable deep model for code functional representation using the huge amount of publicly available task-free source code as well as their textual explanations. And then reuse it for various software defect mining tasks. Experimental results on three major defect mining tasks with real world datasets indicate that by reusing this model in specific tasks, the mining performance outperforms its counterpart that learns deep models from scratch, especially when the training data is insufficient.

Keywords: Software defect mining · Machine learning · Model reuse

1 Introduction

Software Quality Assurance (SQA) is vital in software engineering and one of the biggest influencing factors is *software defects* (also referred as *bugs*). There have been many ways to find bugs, such as conducting software testing. Recently, software defect mining, which leverages data mining techniques to help identifying the software defects, has shown its advantages in reducing the software testing resources, and drawn significant attention.

Various software defect mining tasks can be employed to identify software defects. The major tasks are: *software clone detection*, *defect prediction* and *bug localization*. In software engineering, copy-pasting existing code snippets can usually cause bug propagation. If one code snippet contains a bug, all other

Q. Yang et al. (Eds.): PAKDD 2019, LNAI 11441, pp. 212–224, 2019.
https://doi.org/10.1007/978-3-030-16142-2_17

snippets similar to it may also exist the same bug [13]. Therefore, software clone detection aims to mine such bugs by identifying the cloned code snippets. Apart from that, defect prediction is to directly check if a certain software module contains bugs before a software system releasing, while bug localization refers to locate buggy source code based on bug reports written in natural language submitted by the users after the system releasing.

```
/*Code1: factorial*/
public static void fac(int n)
{
    if(n==0)
        return 1;
    else
        return n*fac(n-1);
}
```

```
/*Code2: factorial*/
public static void fac(int n)
{
    int i,re=1;
    for(i=1;i<=n;i++)
        re=re*i;
    return re;
}
```

```
/*Code3: cumulative sum*/
public static void csum(int n)
{
    int i,re=0;
    for(i=1;i<=m;i++)
        re=re+i;
    return re;
}
```

Fig. 1. An example of three Java code snippets with comments. Code1 and Code2 are similar with the same functionality shown by "factorial" though implemented in different ways (i.e., for-loop and recursion). Code2 and Code3 are dissimilar in functionality shown by "factorial" and "cumulative sum" though nearly the same in appearance.

Many methods have been proposed for these mining tasks. The most common way is to design hand-crafted features for specific mining tasks, such as sequence features, AST (Abstract Syntax Tree) features and PDG (Program Dependence Graph) features in clone detection [1,7,11], software metrics in defect prediction [3,9], bag-of-words features in bug localization [4,21]. Recently, deep neural networks have been applied to tackle software defect mining tasks. Wei and Li [18] address the clone detection problem with deep learning model equipped with AST-based LSTM (Long Short Term Memory) and learning to hash. Huo et al. [6] propose a novel deep model structure based on CNN (Convolutional Neural Network) to learn unified features from both bug reports and source code. They also improve it by taking LSTM to capture the sequential nature of source code [5]. All these deep models have significantly pushed forward the state-of-the-art performance in various software defect mining tasks.

To achieve such promising performance, deep models usually require a huge amount of training data. However, acquiring sufficient number of training data and their labels is usually difficult for software defect mining tasks. For example, after a software system releasing, it takes long time for underlying bugs to be exposed to users for firing bug reports, and hence the number of bug reports that can be used to train the model is small; additionally, much human effort is required to locate the buggy source code from the code bases. Similar problems hold for software clone detection and defect prediction. Therefore, these proposed deep models may not perform as well as they should be in practice.

On the other hand, there has been huge amounts of source code as well as their corresponding textual explanations in the open source software repositories (e.g., SourceForge[1]) and technical forums that discuss and share source code

[1] https://sourceforge.net/.

(e.g., Stack Overflow[2]). These data is publicly available, but is not collected and preprocessed for any particular software mining tasks. One question arises: can we leverage the huge amount of task-free data to help software defect mining tasks with insufficient training data?

Intuitively, if the source code functionality is correctly modeled, it would be apparent to determine whether the code behaves as it is expected to (i.e., whether it contains defects). Thus, the key is to effectively model the functionality of source code which can be reused in many software defect mining tasks to further assist to mine the defect. However, it is sometimes difficult even for software maintenance engineers to determine the source code functionality solely based on the code itself [15], since the same functionality can be implemented in various ways (e.g., summation implemented with for-loop and recursion) and source code similar in appearance may carry different meanings, especially when it is freely written. In this case, additional textual information (e.g., code comments, design documents) may be further referred to. An example of three code snippets with comments is given in Fig. 1 to show how the textual information helps.

In this paper, we propose a novel approach to learn one ReUsable deep Model RUM for the functional representation of source code, which is trained with the huge amount of publicly available source code resources. It is obvious that the code functionality can be well captured with the help of textual information. Unluckily, detailed textual information even comments for source code in specific task is always missing. Therefore, our approach first leverages both source code and their corresponding textual explanations which can be available in public source resources to derive a text-enriched code functionality space. Based on this space, a reusable code functional representation model RUM, which only leverages source code, is constructed by aligning the learned representation towards its counterpart in the text-enriched code functionality space. Such a reusable model can be plugged into different software defect mining tasks with moderate adaptation over the task-specific data to generate the text-enriched functional representations even if no additional textual information available for the specific task. The experimental results on three major software defect mining tasks (i.e., software clone detection, defect prediction and bug localization) with real world datasets indicate that by using this model to generate functional representations for task-specific source code, the mining performance outperforms that learns deep models from scratch, especially when the task-specific training data is insufficient.

2 The Proposed Approach

The goal of the proposed approach is to learn a good code functional representation model using the huge amount of publicly available task-free source code resources and then reuse it for many specific software defect mining tasks.

[2] https://stackoverflow.com/.

Let $\mathcal{O} = \{o_1, o_2, ..., o_N\}$ denotes the code-text set, where $o_i = (c_i, t_i)$, c_i and t_i denote the i-th raw code snippet and corresponding textual comment respectively, N is set size. Let $\mathcal{C} = \{c_1, c_2, ..., c_N\}$ denotes the code set from $\mathcal{O}$.

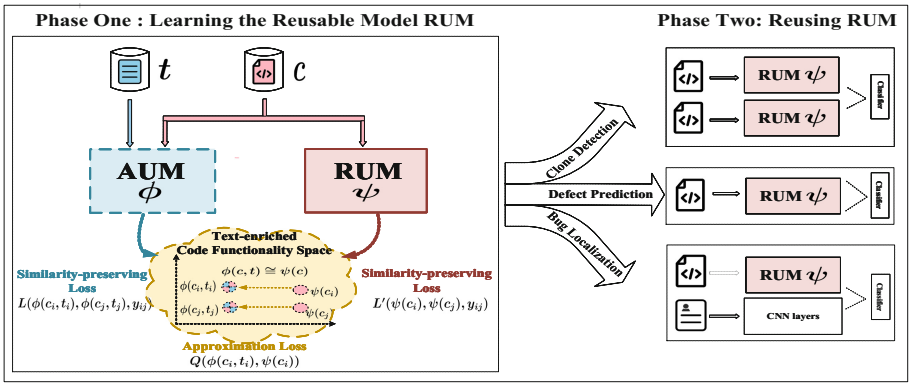

Fig. 2. The framework of learning and reusing RUM, containing two phases. In phase one, we learn a feature mapping ψ in RUM with the help of feature mapping ϕ in AUM. In phase two, we reuse ψ shown in red color in three software defect mining tasks. (Color figure online)

The framework of the proposal approach is shown in Fig. 2. It contains two phases. The first phase shown in the left part is to learn a ReUsable code functional representation Model RUM which accepts source code input. With only code information, semantic functionality is hard to model since the same functionality can be implemented with different lexical or syntactic ways and similar code functionalities are always with similar textual comments (e.g., factorial and cumulative sum shown in Fig. 1). Therefore, we first build an AUxiliary Model AUM to leverage both code and comments to learn the text-enriched code functionality representation space, resulting in the feature mapping $\phi(c, t)$. To further utilize the space, we design a approximation mechanism for RUM with feature mapping $\psi(c, t)$ to align the learned representation to its counterpart, i.e., $\psi(c_i) \cong \phi(c_i, t_i)$, such that it can implicitly encode textual information.

In the second phase shown in the right part, we plug the reusable feature mapping $\psi(c)$ in RUM into different task-specific deep models to replace the code feature extraction substructure, and adapt it towards the task with a small amount of task-specific data. Here, we employ simple fine-tuning technique to RUM in the purpose of verifying the feasibility of our approach. However, any advanced model adaptation techniques can be employed and better performance can be expected. To provide concrete examples on how to reuse RUM in specific defect mining tasks, we select the aforementioned three major software defect mining tasks, namely clone detection, defect prediction and bug localization:

– **RUM for Clone Detection.** Given a source code set $(c_1, ..., c_N)$, the goal is to predict if (c_i, c_j) belong to a clone pair. For this task, double substructures

of $\phi(\cdot)$ are reused for pairwise input. The fully connected layers are followed as the classifier to make a prediction. We denote this model as RUM^{cd}.

- **RUM for Defect Prediction.** Given a source code set $(c_1, ..., c_N)$, the goal is to classify if c_i is defective. For this mining task, we reuse the substructure of $\psi(\cdot)$ and add fully connected layers as classifier. It is denoted as RUM^{dp}.
- **RUM for Bug Localization.** Given a source code set $(c_1, ..., c_M)$ and bug report set $(r_1, ..., r_N)$, the goal is to identify the association y_{ij} between c_i and r_j. For this mining task, we employ the deep model proposed in [6]. Especially, we replace the substructure responsible for source code with the reusable structure $\psi(\cdot)$ in RUM. We denote this model as RUM^{bl}.

It is noteworthy that any software mining tasks, even not for mining defects, may benefit from RUM if they need to model the code functionality. The key of our approach lies in how to derive the reusable text-enriched code functional representation model, i.e., how to learn the feature mapping $\phi(\cdot, \cdot)$ in AUM and the feature mapping $\psi(\cdot)$ in RUM, which will be discussed in the following.

2.1 Auxiliary Model

Auxiliary model AUM is designed to learn a feature mapping $\phi(\cdot, \cdot)$ from $o_i = (c_i, t_i)$ to a text-enriched functionality space where the source code with similar functionality should be mapped close to each other and dissimilar ones should be apart. According to [10], such learning task can be formalized as a binary classification problem that attempts to learn a prediction function $f : \mathcal{O} \times \mathcal{O} \mapsto \mathcal{Y}$. $y_{ij} \in \mathcal{Y} = \{0, 1\}$ indicates whether a pair of input $o_i, o_j \in \mathcal{O}$ is similar or not. Specifically, we employ L_1-distance to weight the affinity of input pairs, and the probability of a pair (o_i, o_j) to be similar can be computed as $f = \sigma(\boldsymbol{\alpha}^{\mathsf{T}} |\phi(o_i) - \phi(o_j)|)$, where σ is the sigmoid activation function and $\boldsymbol{\alpha}$ is the parameter to be learned. We solve the learning problem by optimizing the following regularized similarity-preserving loss function:

$$\min_{f} \quad \mathcal{L} + \lambda \Omega(f), \tag{1}$$

where $\mathcal{L}$ is the cross-entropy loss, $\Omega(f)$ is the L_2 regularization term and λ is the trade-off parameter. This objective can be effectively optimized using SGD.

Note that the number of source code with similar functionality is usually far less than that with dissimilar functionality. Such imbalanced distribution may severely affect the quality of learned code functionality space. To reduce the influence, we impose a larger cost for miss-classifying the similar code pairs (denoted by cost_{fn}) and a smaller cost for miss-classifying the dissimilar code pairs (denoted by cost_{fp}). Therefore, $\mathcal{L}$ can be defined as:

$$\mathcal{L} = \sum_{i,j} (\text{cost}_{fp}(1 - y_{ij}) \log(1 - f(o_i, o_j) + \text{cost}_{fn} y_{ij} \log f(o_i, o_j)). \tag{2}$$

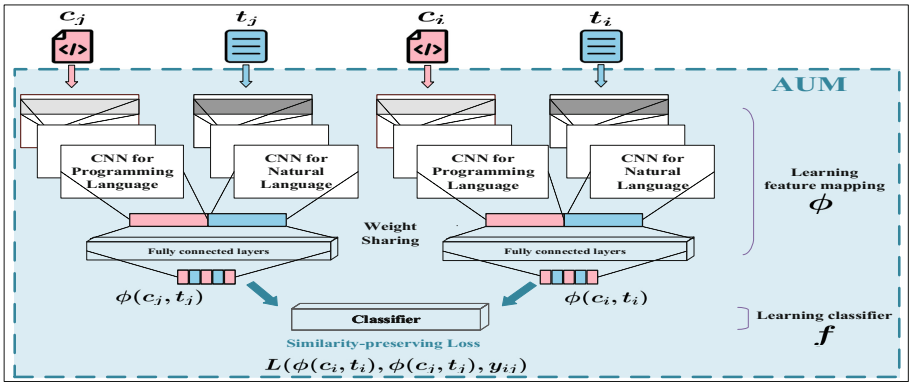

Fig. 3. The siamese structure of AUM than is weighted sharing for (c_i, t_i) and (c_j, t_j).

We instantiate the auxiliary network with siamese convolutional neural network. Siamese network [2], consisting of two identical neural networks with their weights tied, is usually employed to differentiate the paired input data points [10,12]. Thus, we leverage the siamese structure to help modeling the similarity of source code pair. The network structure of AUM is shown in Fig. 3. Source code is always written in programming language in which multiple continues statements is constructed in a block to convey the information, e.g., for-loops and while-loops. While text is written in natural language in a flat way that several words together can express the complete meanings. Thus, we design two different feature extraction modules for each in AUM. In the code feature extraction layers, since convolutional neural networks have shown great performance in [6], we use the same convolution layers with specific convolution operations for code structure to extract the semantic features for source code. In the text feature extraction layers, we use the standard approach in [8] to extract the text features. Next, we use fully connected layers to further fuse the code features and text features and get final representations. The above feature extraction layers are weight-shared for a pair of input (c_i, t_i) and (c_j, t_j) to get unbiased representations. In the end, fully connected layers followed by a sigmoid layer are constructed to build a classifier based on fused representations for optimization objective (i.e., similarity-preserving loss).

2.2 Reusable Model

Reusable model RUM aims to learn a feature mapping $\psi(\cdot)$ from c_i to the same text-enriched functionality space by aligning it to its text-enriched counterpart $o_i = (c_i, t_i)$. To achieve it, we force $\psi(c_i) \cong \phi(c_i, t_i)$ by imposing approximation loss over the distances between $\psi(c_i)$ and $\phi(c_i, t_i)$, as defined in Eq. (3):

$$\mathcal{Q} = \sum_i ||\psi(c_i) - \phi(c_i, t_i)||_2^2. \tag{3}$$

By minimizing Eq. (3), we can squeeze $\psi(c_i)$ into the neighborhood of $\phi(c_i, t_i)$ from all directions in the text-enriched space. However, when some hard cases occur, it is difficult to push some $\psi(c_i)$ towards $\phi(c_i, t_i)$, it may end up with a relatively large neighborhood. In this case, for some similar code pairs c_i and c_j, even if $\psi(c_i)$ and $\psi(c_j)$ may be squeezed into the neighborhoods of $\phi(c_i, t_i)$ and $\phi(c_j, t_j)$, they may still be distant by mapping them from the opposite direction of their counterparts. To overcome it, we also impose a similarity-preserving loss over source code pairs c_i and c_j, as defined in Eq. (4):

$$\mathcal{L}' = \sum_{i,j} (\text{cost}_{fp}(1 - y_{ij}) \log(1 - g(c_i, c_j)) + \text{cost}_{fn} y_{ij} \log g(c_i, c_j)). \tag{4}$$

Therefore, we solve the problem of learning the reusable model RUM by optimizing the following regularized objective loss function,

$$\min_{g} \quad \mathcal{Q} + \beta \mathcal{L}' + \lambda \Omega(g), \tag{5}$$

where $\Omega(g)$ is the L_2 regularization term, β and λ' are the trade-off parameters.

We instantiate this learning task also by siamese convolutional neural network. The network structure of RUM is the same as that of AUM except for the text feature extraction layers since RUM only takes source code as the input. Similar to AUM, the feature extraction layers are weight-shared and a classifier is built based on the approximation representations for optimization objective.

3 Experiment

In the experiment, we first show how good the functional representation learned by RUM is and then we show the benefit of reusing RUM for various software defect mining tasks.

3.1 How Good Is RUM

In this section, we conduct experiments on the real-world dataset *Stack Overflow* downloaded from Stack Exchange[3] to evaluate the performance of identifying functional similar code pairs based on the learned representations by RUM.

The dataset contains 8237 questions (text) and 8237 answers (code), in which each question is along with a answer. In order to get the similarity label, we label dual problems that are with similar question as similar pairs and generate dissimilar pairs from non-dual problems, which totally get 16839 pairs.

Since RUM is benefit from the text-enriched functionality space, we compare RUM with RSiaCNN which learns functional representations only from code. In both, the network parameters are chosen as follow: the convolution filter size in code feature extraction layers is 3, 4 with 100 feature maps each and in text

[3] https://archive.org/details/stackexchange.

feature extraction layers is 2, 3 with 100 feature maps each. We set experiment dropout probability $p = 0.5$ and activation function $\text{ReLU}(x) = \max(x, 0)$.

The performance ratio of RSiaCNN and RUM over AUM in terms of AUC is 91.4% and 95.4%, respectively. Thus, the performance can be improved by nearly 4% if trained with the help of text-enriched space. The similar conclusion can be observed in terms of F1 which improved by 6%. Next, we will verify the effectiveness of RUM on three major defect mining tasks.

3.2 Reusing RUM for Clone Detection

BigCloneBench [16] is a widely used benchmark dataset with known true and false clones. Following [18], we extract 6282 code snippets as dataset. Since *BigCloneBench* is highly imbalanced, we measure the performance in terms of AUC, F1 and Recall. Besides, Top k Rank ($k = 10$) is recorded to measure the retrieval performance. We first compare RUM^{cd} to the state-of-the-art deep models DeepClone [19] and CDLH [18]. Further more, we compare with our variants SiaCNN, which is with the same structure as RSiaCNN but trained from scratch, and RSiaCNN^{cd} to evaluate the effective of enriched text information.

Table 1. Top 10 Rank and AUC of all methods on training data with different sizes.

Methods	Top 10 Rank				AUC			
	50	100	250	500	50	100	250	500
DeepClone	.894 ∘	.894 ∘	.894 ∘	.894 ∘	.483 ∘	.483 ∘	.483 ∘	.483 ∘
CDLH	.795 ∘	.774 ∘	.794 ∘	.858 ∘	.500 ∘	.500 ∘	.500 ∘	.500 ∘
SiaCNN	.659 ∘	.698 ∘	.753 ∘	.852	.507 ∘	.533 ∘	.650 ∘	.757 ∘
RSiaCNN^{cd}	.808 ∘	.866 ∘	.876 ∘	.905	.609 ∘	.665	.781	.829
RUM^{cd}	**.912**	**.917**	**.933**	**.935**	**.621**	**.666**	**.794**	**.838**

We randomly sample 5000 code for training and 500 code for testing, resulting in 25000000 (5000×5000) training pairs and 250000 (500×5000) test pairs. To evaluate the performance on small datasets, we use only small sizes of the train pairs, assuming N ($N = 50, 100, 250, 500, 750, 1000$ respectively), to train and test on all test pairs which is large enough to prove our performance.

All experiments are randomly repeated 30 times and we report the average results. The performance with respect to Top k Rank and AUC of all methods on different training samples are tabulated in Table 1 where the best performance on each dataset is boldfaced. The performance with respect to F1 score is depicted in Fig. 4. We conduct Pairwise t-test at 95% confidence level. The compared methods that are significant inferior than our approach will be marked with "∘" and significant better will be marked with "•".

From the results in Table 1, we can observe that when training size is very small, e.g. 100, RUM^{cd} can achieve the best performance (0.917) in terms of

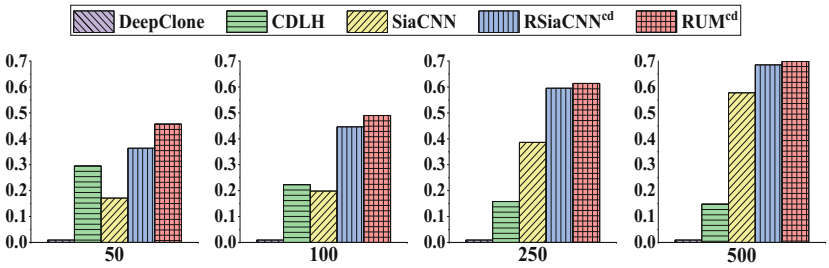

Fig. 4. F1 score of all methods on training data with different size.

Top 10 Rank which improves CDLH (0.774) by 14.3% and SiaCNN (0.698) by 21.9% since they are easy to overfit. When compared to the unsupervised method DeepClone which trained with all code, we still get 1.8% improvement with only 50 training samples. Similar traces can be found in Fig. 4. The superiority of RUM is obvious. We further evaluate effectiveness of text information. Indicted in Table 1, the performance of RUM^{cd} is better than RSiaCNN^{cd} and improves by 3%–9% in terms of Top 10 Rank. It shows that encoded text information is beneficial when the code information is not enough. Taking a concrete example of the cloned pair Code1 and Code2 in Fig. 1, we can get the similar enriched representations by using RUM with the help of comments "factorial" though they are very dissimilar in lexical structure.

3.3 Reusing RUM for Defect Prediction

For defect prediction task, we conduct experiments on the widely used benchmark datasets [22]. It contains source code files and detailed software metric information, such as complexity metrics (e.g., number of methods calls, total lines of code), structure of AST and so on. It also gives the number of defects that are reported in the first six months before and after releasing, named as pre-release defects and post-release defects respectively. In this experiment, we use three different projects as our datasets and use the number of post-release defects as prediction label. The statistics of the datasets can be found in Table 2.

As indicated in Table 2, the number of defective source code is very imbalanced. Therefore, we use AUC to evaluate the performance. We compare RUM^{dp} with a baseline Logistic Regression LR and two state-of-the-art methods

Table 2. Statistics of three datasets in defect prediction.

Datasets	# attributes	# instances	# defective
Debug	198	194	13%
UI	198	1166	4%
SWT	198	841	17%

DBN [20] and AST-DBN [17]. Besides, we compare with two variants RSiaCNN^{dp} and P-CNN [6] which is with the same code feature extraction structure as RUM but trained from scratch. For each dataset, we randomly sample 30% data to train and the remaining data to test. All experiments are randomly repeated 30 times and the average results are reported in Fig. 5.

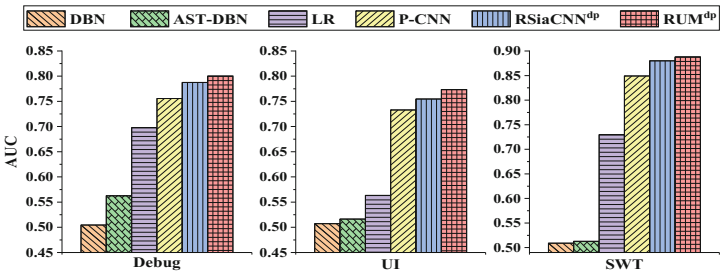

Fig. 5. AUC of compared methods on all datasets in defect prediction.

It can be observed from Fig. 5 that RUM^{dp} achieves the best average performance (0.821) among all compared approaches. Compared with LR trained with software metric features (0.664), RUM^{dp} can improve the average performance by 15.7%. When compared with deep models DBN (0.507) and AST-DBN (0.530), RUM^{dp} can also improve by 31.4% and 29.0%. It is notable that the performance of DBN and AST-DBN is even worse than LR, we explain that DBN extracts code features in an unsupervised way. To evaluate the effectiveness of reusable code functional representations, we use P-CNN for comparison. It is clearly that RUM^{dp} can improve the performance of P-CNN (0.780) by 4.1%. Also, we compare RUM^{dp} with RSiaCNN^{dp} to evaluate the effectiveness of encoded text information. Though RUM^{dp} is fine-tuned without any text input, it can still improve RSiaCNN^{dp} (0.807) by 1.4% on average. Therefore, the encoded text information is useful for finding more defective modules. Here we give a more intuitive explanation for the usefulness of text. If one aims to get factorial function but wrongly writes Code3 in Fig. 1, the encoded text information "cumulative summation" in the reusable representation can help the detection.

3.4 Reusing RUM for Bug Localization

In bug localization, we extract different well-known open source software projects and the ground truth of relevance of bug reports and source files using bug tracking system (Bugzilla) and version control system (Git), following [6]. We use matched code-report pair as positive instance. To generate the negative instance, we label the reports with irrelevant code files as negative. Table 3 shows the detailed information of used datasets.

We use Top k Rank ($k = 10$) to measure our performance, which has been widely applied for evaluation in information retrieval based bug localization

Table 3. Statistics of three datasets in bug localization.

Datasets	#source files	#bug reports	#total matches
Debug	249	132	301
UI	1152	314	698
JDT	1980	1005	1610

problems [14,21]. We compare our method with the state-of-the-art methods NP-CNN [6], LSTM-CNN [5] and RSiaCNN^{bl}. For each dataset, we randomly sample 30% data to train our model, and test on the remaining data. All experiments are randomly repeated 30 times and we report the average results. The performance is depicted in Fig. 6.

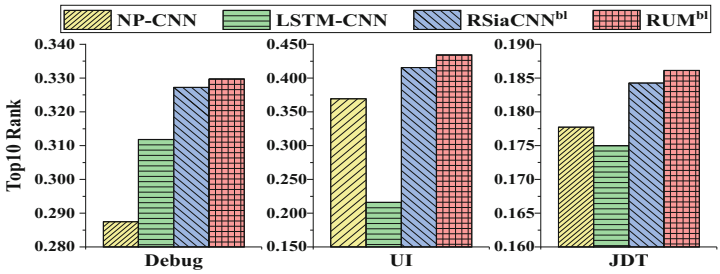

Fig. 6. Top 10 Rank of compared methods on all datasets in bug localization.

Figure 6 indicates that RUM^{bl} achieves the best average Top k Rank at 0.317, which improves the average performance of NP-CNN (0.279) by 3.8% and LSTM-CNN (0.234) by 8.3%. It should be notable that LSTM-CNN performs better than NP-CNN when the number of training samples is small, we explain that LSTM without any dropout layers is more easy to overfit. Further to evaluate the effectiveness of text information, we use RSiaCNN^{bl} for comparison, which is pretrained with only source code. It can be observed that RUM^{bl} improves RSiaCNN^{bl} (0.309) by 0.8% on average in terms of Top k Rank, indicating that encoded text information is also helpful to locate buggy source files. For a more clear explanation, assuming that we are given the bug report "I always get the same value for factorial of n" and one aims to locate some factorial function containing bugs, then the encoded text "factorial" in code representations is consistent to report description "factorial" and is useful in localization.

4 Conclusion

In this paper, we propose a novel framework to learn one deep model for the code functional representation using the huge amount of publicly available task-free

source code and their textual comments, and reuse it for many software defect mining tasks. Experimental results on three major software defect mining tasks indicate that by reusing this model in specific task, the mining performance outperforms its counterpart that learns deep models from scratch, especially when the task-specific training data is insufficient.

Acknowledgment. This research was supported by National Key Research and Development Program (2017YFB1001903) and NSFC (61751306).

References

1. Alemi, M., Haghighi, H., Shahrivari, S.: CCFinder: using Spark to find clustering coefficient in big graphs. J. Supercomput. **73**(11), 4683–4710 (2017)
2. Bromley, J., Guyon, I., LeCun, Y., Säckinger, E., Shah, R.: Signature verification using a Siamese time delay neural network. In: Advances in Neural Information Processing Systems, pp. 737–744 (1993)
3. D'Ambros, M., Lanza, M., Robbes, R.: Evaluating defect prediction approaches: a benchmark and an extensive comparison. Empir. Softw. Eng. **17**(4–5), 531–577 (2012)
4. Gay, G., Haiduc, S., Marcus, A., Menzies, T.: On the use of relevance feedback in IR-based concept location. In: Proceedings of the 25th IEEE International Conference on Software Maintenance, pp. 351–360 (2009)
5. Huo, X., Li, M.: Enhancing the unified features to locate buggy files by exploiting the sequential nature of source code. In: Proceedings of the 26th International Joint Conference on Artificial Intelligence, pp. 1909–1915 (2017)
6. Huo, X., Li, M., Zhou, Z.H.: Learning unified features from natural and programming languages for locating buggy source code. In: Proceedings of the 25th International Joint Conference on Artificial Intelligence, pp. 1606–1612 (2016)
7. Jiang, L., Misherghi, G., Su, Z., Glondu, S.: DECKARD: scalable and accurate tree-based detection of code clones. In: Proceedings of the 29th International Conference on Software Engineering, pp. 96–105 (2007)
8. Johnson, R., Zhang, T.: Effective use of word order for text categorization with convolutional neural networks. In: Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pp. 103–112 (2015)
9. Kim, S., Zimmermann, T., Whitehead Jr., E.J., Zeller, A.: Predicting faults from cached history. In: Proceedings of the 29th International Conference on Software Engineering, pp. 489–498 (2007)
10. Koch, G., Zemel, R., Salakhutdinov, R.: Siamese neural networks for one-shot image recognition. In: Proceedings of the 32nd International Conference on Machine Learning Deep Learning Workshop, vol. 2 (2015)
11. Komondoor, R., Horwitz, S.: Using slicing to identify duplication in source code. In: Cousot, P. (ed.) SAS 2001. LNCS, vol. 2126, pp. 40–56. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-47764-0_3
12. Mueller, J., Thyagarajan, A.: Siamese recurrent architectures for learning sentence similarity. In: Proceedings of the 30th AAAI Conference on Artificial Intelligence, pp. 2786–2792 (2016)
13. Roy, C.K., Cordy, J.R.: A survey on software clone detection research. Queen's Sch. Comput. TR **541**(115), 64–68 (2007)

14. Saha, R.K., Lease, M., Khurshid, S., Perry, D.E.: Improving bug localization using structured information retrieval. In: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering, pp. 345–355 (2013)
15. de Souza, S.C.B., Anquetil, N., de Oliveira, K.M.: A study of the documentation essential to software maintenance. In: Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting & Designing for Pervasive Information, pp. 68–75 (2005)
16. Svajlenko, J., Islam, J.F., Keivanloo, I., Roy, C.K., Mia, M.M.: Towards a big data curated benchmark of inter-project code clones. In: Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution, pp. 476–480 (2014)
17. Wang, S., Liu, T., Tan, L.: Automatically learning semantic features for defect prediction. In: Proceedings of the 38th International Conference on Software Engineering, pp. 297–308 (2016)
18. Wei, H.H., Li, M.: Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In: Proceedings of the 26th International Joint Conference on Artificial Intelligence, pp. 3034–3040 (2017)
19. White, M., Tufano, M., Vendome, C., Poshyvanyk, D.: Deep learning code fragments for code clone detection. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, pp. 87–98 (2016)
20. Yang, X., Lo, D., Xia, X., Zhang, Y., Sun, J.: Deep learning for just-in-time defect prediction. In: Proceedings of the 2015 IEEE International Conference on Software Quality, Reliability and Security, pp. 17–26 (2015)
21. Zhou, J., Zhang, H., Lo, D.: Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. In: Proceedings of the 34th International Conference on Software Engineering, pp. 14–24 (2012)
22. Zimmermann, T., Premraj, R., Zeller, A.: Predicting defects for Eclipse. In: Proceedings of the 3rd International Workshop on Predictor Models in Software Engineering, p. 9 (2007)