Learning Semantic Features for Software Defect Prediction by Code Comments Embedding*

Xuan Huo*, **Yang Yang***, **Ming Li***† and **De-Chuan Zhan***†
*National Key Laboratory for Novel Software Technology, Nanjing University
†Collaborative Innovation Center of Novel Software Technology and Industrialization
Nanjing 210023, China
{huox, yangy, lim, zhandc}@lamda.nju.edu.cn

***Abstract*—Software Quality Assurance (SQA) is essential in software development and many defect prediction methods based on machine learning have been proposed to identify defective modules. However, most existing defect prediction models do not provide good defect prediction results, and the semantic features reflecting the detective patterns may not be well-captured via traditional feature extraction methods. More information such as code comments should be also be embedded to generate semantic features respecting the source code functionality. Therefore, how to embed code comments for defect prediction is a big challenge, and another problem is that many comments of source code are missing in real-world applications.**

In this paper, we propose a novel defect prediction model named CAP-CNN (Convolutional Neural Network for Comments Augmented Programs), which is a deep learning model that automatically embeds code comments in generating semantic features from the source code for software defect prediction. To overcome the missing comments problem, a novel training strategy is used in CAP-CNN that the network encodes and absorb comments information to generate semantic features automatically during training process, which does not need testing modules to contain comments. Experimental results on several widely-used software data sets indicate that the comment features are able to improve defect prediction performance.

***Index Terms*—data mining, software mining, software defect prediction**

I. Introduction

Technical advances in software systems have been researched widely during recent years and they become increasingly versatile and powerful. Software Quality Assurance (SQA), which is vital to the success of software projects, has drawn much attention from developers and researchers. However, software quality assurance usually involves coding reviews and extensive testing of software modules, which is resources and time consuming. To make SQA cost-effective, software defect prediction has been applied to automatically identify defect-prone software modules of the software, in order to guide the SQA resource allocation to those modules which are most likely defect-prone.

To automatically identify defective modules, many defect prediction models based on machine learning has been widely studied in the past years [1]–[8]. The classification model is usually trained based on a collection of software modules whose defect-proneness are known, and this model can be further used to classify software modules into *defective* or *clean*. One of the most important factors that affect the defect prediction performance is the feature representation of the source code. In previous studies, researchers have designed many software metrics to represent the features of the source code for defect prediction, e.g., Menzies et al. [1] utilized some static features to build defect prediction models. Zimmermann and Nagappan [9] extracted features from the structural measurement of dependency graphs. Since different developer may have different coding style, Jiang et al. [10] extracted developer features and built prediction model separately for each developer to identify defective modules. In addition, Wang et al. [8] proposed a deep learning model based on Deep Belief Network (DBN) that can automatically learn semantic features from the Abstract Syntax Tree. The DBN model shows good performances in both Within-Project Defect Prediction and Cross-Project Defect Prediction.

However, since the structural semantics and property of source code is different from natural language, the characteristics of the defective patterns may not be well-captured by traditional defect prediction features. Noticing that some defects in modules may not be caused by the error of the code structure, but are caused by the error of code functionality. For example, Figure 1 shows an example of a defective module in project Apache and its corresponding defects description extracted from the bug report. The description of the defect shows that the connections to smtp server have no authentication and the `mail.smtp.auth` is always set to true. If we only consider the structure and the syntax of source code, it is hard to find the defect because the syntax of this module is correct. To overcome this problem, it is important to generate semantic feature representation that reflecting the structure and functionality of source code. Traditional features for defect prediction do not reflect the semantics of source code.

Considering the format of source code, it can be found that not only can the code functionality be extracted from the source code, but also can be generated from code comments. Source code comments are existed along with source code, which are some text in natural language that describes the functionality of the code. Comments can be regarded as another view of the source code to help generate semantic features reflecting the code functionality for identifying de-

*This research is supported by National Key Research and Development Program (2017YFB1001903) and NSFC(61773198, 61632004).

978-1-5386-9159-5/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDM.2018.00133

fective modules. Unfortunately, in many cases that the source code from archives may not contain comments. For example, some developers do not have the habit to write comments, or the project is in the early stage of development. Therefore, how to learn semantic features from the source code based on comments embedding is a big challenge, and another problem is that how to deal with missing comments situations.

To overcome this problem, we propose a novel defect prediction model named CAP-CNN (Convolutional Neural Network for Comments Augmented Programs), which is able to embed code comments to automatically generate semantic features from the source code for defect prediction. The structure and functionality of source code are reflected by the generated semantic features, which may improve software defect prediction performance. To deal with missing comments situation, the model employs a novel learning strategy to encode code comments into semantic feature during the training process, and do not need testing modules to contain comments for defect prediction. Experimental results on widely-used software defect prediction data sets indicate that code comments can help improve defect prediction performance, and the CAP-CNN model performs better than previous state-of-the-art methods for both Within-Project Defect Prediction and Cross-Project Defect Prediction.

The contributions of our work can be summarized:

- We are the first to embed code comments to extract semantic features from the source code for software defect prediction, generating a better feature representation reflecting the structure and functionality of source code.
- We propose a novel defect prediction model named CAP-CNN, which takes raw modules as input and generates semantic features of source code by embedding code comments for identifying software defects.
- We employ a novel learning strategy to encode the code comments into semantic feature extraction during the training process, which helps to deal with missing comments situations.

The rest of the paper is structured as follows: Some related work for defect prediction and deep learning models on software engineering are reviewed in Section 2. Section 3 discusses the details of our approach. Section 4 reports the empirical studies and discussions. Finally, Section 7 concludes the work and discusses some future work.

II. Related Work

Over the years, many prediction models have been proposed to predict defects of software modules [8], [10]–[14]. Most techniques leverage software features from historical data to build classification models based on machine learning to predict defective modules. Since the feature metrics of software modules play a very import role in defect prediction, many researchers have studied the performance of applying different metrics. Based on these features, several work considers defect prediction from practical aspect recent years. Li et al. [5] proposed a sample-based defect prediction model based on active and semi-supervised learning to alleviate the burden

Source code:
MailConfiguration.java:

```
/*
 Represents the configuration data for communicating over email
*/
......
private Properties createJavaMailProperties() {
// clone the system properties and set the java mail properties
+     if (username != null) {
        properties.put("mail." + protocol + ".user", username);
        properties.put("mail.user", username);
+       properties.put("mail." + protocol + ".auth", "true"); }
+   else {
+       properties.put("mail." + protocol + ".auth", "false");
+   }
    properties.put("mail." + protocol + ".rsetbeforequit", "true");
-   properties.put("mail." + protocol + ".auth", "true");
```

The description of the defect: Connections to smtp servers with no authentication fail. The mail.smtp.auth is always set to true. The path sets it to false if no username is given.

Figure 1. An example of a defect module and its corresponding bug report from project Apache. It can be observed that the comments from the source code describes its function and can be used for identifying defects.

of collecting many defective labels. In order to overcome the distribution difference between different projects, Nam et al. [15] proposed Transfer Component Analysis (TCA) for Cross-Project Defect Prediction (CPDP). Jing et al. [7] proposed a cost-sensitive discriminative dictionary learning (CDDL) approach for defect prediction.

In recent years, some deep learning models have been studied on software engineering issues [16]–[19]. For example, Yang et al. [16] proposed an approach that leveraged deep learning to generate features from 14 traditional change level features and then used these new features to predict whether a commit is buggy or not. In order to bridge the gap between programs' semantics and defect prediction features, Wang et al. [8] leveraged a model based on Deep Belief Network (DBN) to learn semantic features from Abstract Syntax Trees (ASTs). Their evaluation on ten open source projects shows that the automatic learning semantic features could improve both Within-Project and Cross-Project Defect Prediction. Mou et al. [17] used deep learning to model programs and showed that deep learning can capture programs' structural information. Considering additional semantics beyond the lexical terms of source code in programming languages, Huo et al. [18], [20] proposed a particular convolutional neural network to learn unified features from source files in programming language and bug reports in natural language for locating buggy source code.

III. Approach

In this section, we describe the framework of our proposed model named CAP-CNN (Convolutional Neural Network for Commets Augmented Programs) for software defect prediction in details. Before introducing our model, some notations are firstly presented. Suppose we have n modules, $\mathcal{X} = \{\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_n\}$ denotes the set of modules in source archives. The code and comments are both considered in our work, so we have $\mathbf{x}_i = (\mathbf{x}_i^c, \mathbf{x}_i^m)$, where $\mathbf{x}_i^c$ and $\mathbf{x}_i^m$ indicates

source code and comments, respectively. Software defect prediction can be formalized as a learning task, which attempts to learn a prediction function $F : \mathcal{X} \mapsto \mathcal{Y}$. $y_i \in \mathcal{Y} = \{1, 0\}$ indicates whether module i contains defect.

A. The General Framework

The general process of software defect prediction of our work is illustrated in Figure 2. Noticing that the training process and testing process is different. During the training process (Figure 2-(a)), source modules are firstly split into code and comments part, which are then encoded as vector representations, respectively. CAP-CNN then utilizes two convolutional neural networks to extract semantic features from comments and building the classification model. The classifier (CAP-CNN) is trained to learn the semantic features. However, testing modules do not contain comments, therefore, during testing process (Figure 2-(b)), only code part is fed into the trained model for defect prediction.

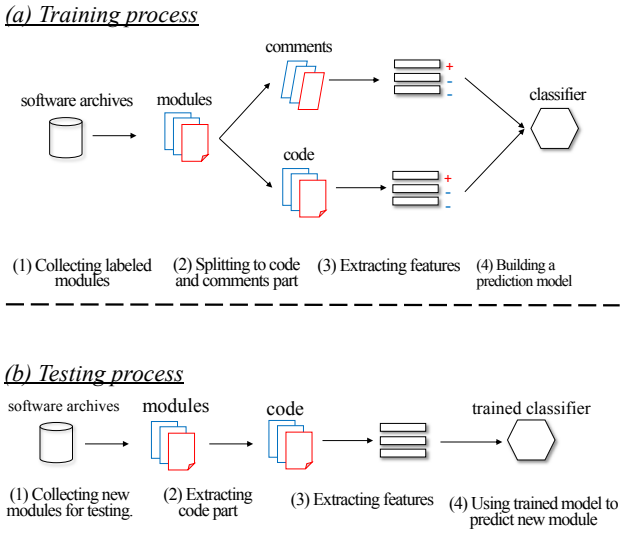
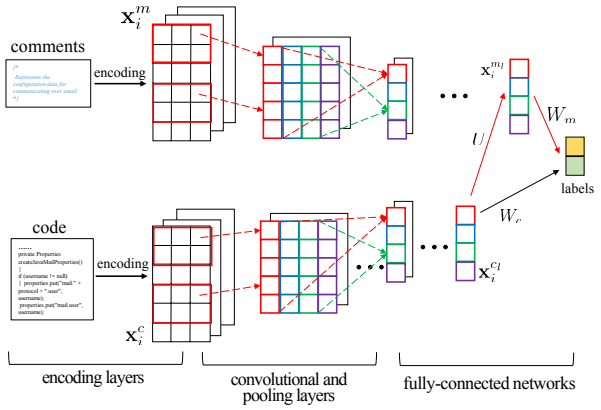


Figure 2. The general process of CAP-CNN for software defect prediction. The above figure shows training process and the below figure shows testing process.

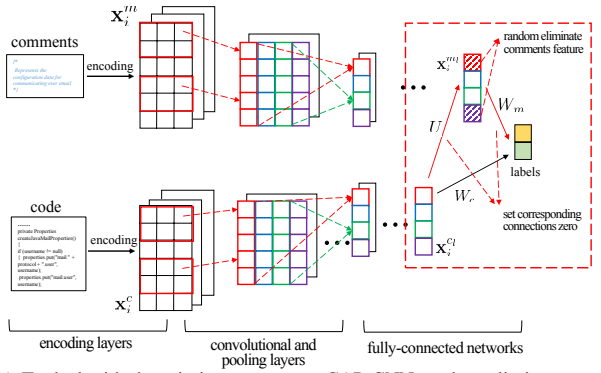
B. CNN for Comments Augmented Programs

In this section, we introduce some details about the structure and the training process of CAP-CNN. Firstly, raw code and comments should be encoded as feature vectors via pre-trained word2vec model [21] before being fed into the deep mode. Then, since source code in programming language and comments in natural language may have different structure and property, two convolutional neural networks are employed to process source code and comments separately.

To generate semantic comments features, we followed similar convolutional neural network defined as CNN^m in [22], which takes raw feature vectors encoded from comments as input and the output of last layer $\mathbf{x}_i^{m_l} \in \mathbb{R}^m$ is the generated semantic feature of source comments, where m is the number of filters.



(a) The general process of extracting semantic features by CAP-CNN for defect prediciton.



(b) To deal with the missing comments, CAP-CNN random eliminates m_e comments features without replacement and set corresponding connections in U and W_m to zero when training each batch of module.

Figure 3. The CAP-CNN model for software defect prediction with missing comments.

The source code is different from comments since they are different languages and may contain different structural semantics, so a different convolutional network named CNN^c is used for code feature extraction. The structure of CNN^c is similar in [18] that two layers of convolutional and pooling layers are utilized to generate high-level semantic feature. The first convolutional and pooling layer is based on token-level, where filters are slided within statement and extract the features from tokens, in this way the integrity of each statement will be carefully preserved. The subsequent convolutional and pooling layers is based on statement-level, where convolutional filters are employed to model the interactions between statements. To maintain consistency with semantic comments features, the number of filters are also set as m. Finally, the high-level feature denoted as $\mathbf{x}_i^{c_l} \in \mathbb{R}^m$ is generated to represent the source code.

Specifically, the code and comments of the module $\mathbf{x}_i$ can be finally extracted and represented as $\mathbf{x}_i^{c_l}$ and $\mathbf{x}_i^{m_l}$ via multiple layers. Eventually two fully are connected to the output labels y_i with a nonlinear softmax function. The parameters of these two fully-connected network can be denoted as W_c and

Algorithm 1 Comments embedding based Programming Convolutional Neural Network for defect prediction

Input:

the set of training modules $\mathcal{X}$; the set of test modules $\mathcal{X}_t$; eliminate number of features in each epoch n_e; batch size n_b; max-iteration n_i;

Process:

1: Separate training module $\mathcal{X}$ into code sets $\mathcal{X}^c$ and comments sets $\mathcal{X}^m$.
2: Encode code and comments into vector representation $(\mathbf{x}_i^c, \mathbf{x}_i^m)$, respectively.
3: **repeat**
4: Random pick up n_b modules from $\mathcal{X} = (\mathbf{x}_i^m, \mathbf{x}_i^c)$ without replacement to create training batch.
5: Forward propagate via network and calculate the training loss $\mathcal{L}$ by Eq.(1).
6: Calculate the derivative $\partial\mathcal{L}/\partial W_c$, $\partial\mathcal{L}/\partial W_m$, $\partial\mathcal{L}/\partial U$, $\partial\mathcal{L}/\partial\Theta$. Back propagate and update the network parameters W_m, W_c, U, Θ.
7: Randomly eliminate n_e comments features without replacement and set corresponding connections of U and W_m to zero;
8: **until** the max-iteration is reached or convergence
9: Encode test modules j as vectors representations and use trained model CAP-CNN to predict test modules;

Output:

P_j, which indicates the prediction of each test module j indicating defective or clean.

W_m and the parameters of the convolutional layers can be denoted as $\Theta = \{\theta_1, \theta_2, \ldots, \theta_l\}$. Besides, there is also linear connection between the generated semantic code features and comments features, which is also a fully connected structure and can be denoted as U. Therefore, the loss function implied in CAP-CNN is:

$$\mathcal{L}(\Theta, W_m, W_c, U) = \sum_{i=1}^{N} \ell(\mathbf{x}_i^c, \mathbf{x}_i^m, y_i) + \lambda L_r \qquad (1)$$

where

$$\begin{aligned} \ell(\mathbf{x}_i^c, \mathbf{x}_i^m, y_i) &= \hat{\ell}(\mathbf{x}_i^{c_l}, \mathbf{x}_i^{m_l}, y_i) + \tilde{\ell}(\mathbf{x}_i^{c_l}, \mathbf{x}_i^{m_l}) \\ \hat{\ell}(\mathbf{x}_i^{c_l}, \mathbf{x}_i^{m_l}, y_i) &= \ell_c(\mathbf{x}_i^{c_l}, y_i; W_c) + \ell_c(\mathbf{x}_i^{m_l}, y_i; W_m) \\ \tilde{\ell}(\mathbf{x}_i^{c_l}, \mathbf{x}_i^{m_l}) &= \frac{1}{2}||\mathbf{x}_i^{c_l} - \mathbf{x}_i^{m_l} U||_F^2 \end{aligned}$$

Here $\mathbf{x}_i^{c_l}$ and $\mathbf{x}_i^{m_l}$ are the output of the last layer of convolutional network according to the input module $\mathbf{x}_i$. $\hat{\ell}(\mathbf{x}_i^{c_l}, \mathbf{x}_i^{m_l}, y_i)$ is the label prediction loss function, which consists of two part $\ell_c(\mathbf{x}_i^{c_l}, y_i; W_c)$ and $\ell_c(\mathbf{x}_i^{c_m}, y_i; W_m)$, indicating the loss using semantic code feature and semantic comments feature for prediction, respectively. $\tilde{\ell}(\mathbf{x}_i^{c_l}, \mathbf{x}_i^{m_l})$ is the loss function that measures the difference between the generated semantic comments and code feature. L_r is the regularization term and the parameter λ controls the trade-off between the loss and regularization.

However, in real-word applications, some source code do not contain code comments due to various reasons . For example, some developers are not used to write comments, or the project are at the primary stage during development process. In these situations, only code comments are available in the training data sets and the comments are missing during defect prediction testing process, which is a big challenge. To overcome this problem, we propose a novel learning strategy during CAP-CNN training process, which aims to push the "information" from comments features to the deep model to learn a better semantic feature representation. Our model is able to deal with the situation that the training modules contain both code and comments part but the testing module does not contain comments.

The training progress of CAP-CNN to deal with missing comments testing modules is illustrated in Figure 4-(b). Since testing modules do not contain comments, we need to eliminate the influence of comments feature $\mathbf{x}_i^m$ during the training process, and remove those connected parts corresponding to feature $\mathbf{x}_i^m$ gradually and finally vanish all related components. Besides, in each adjustment of removing parts of those components, it requires additional steps for making the whole deep structure self-consistent which can further reduce the prediction errors. We propose a novel learning strategy in CAP-CNN: when CAP-CNN is trained with each small batch of modules, the model randomly removes several components of features and the connections to these components are also set to zero. In each iteration, the step randomly eliminates components without replacement. After eliminating the comments features, the calculated loss is back-propagated again to make the deep structure self-consistent. Finally, after all batches of training models are fed into the model for processing, this elimination process will cause all the comments features $\mathbf{x}^m$ disconnected and all the connections connected to comments features are vanishing. After all comments are removed, the trained model only takes code as input. Thus, in the testing phase, only source code features $\mathbf{x}^c$ is required as inputs for the deep model and no comments features are further desired. Details are shown in Algorithm 1.

IV. Experiments

To evaluate the effectiveness of our proposed CAP-CNN model, we conduct experiments on open source projects and compare it with state-of-the-art defect prediction models.

A. Experiments settings

The benchmarks used in our experiments are available from PROMISE [1] [23], which is an open data set and have been used in many previous studies [8], [23], [24]. Each data set corresponds to one software project in Apache. They collect bug information and analyze the logs from the source code repository (SVN or CVS) to decide whether a commit is a bug fix. The data sets used in our experiments are *camel*, *ivy*, *log4j*, *lucene*, *poi*, *synapse*, *xalan* and *xerces*. More details about the data sets can be referred in [23].

[1] http://openscience.us/repo/defect

For Within-Project Defect Prediction, we compare several state-of-the-art traditional classification models: Logistic Regression (LR), Naive Bayes (NB) and ADTree, which are widely used and shown their good performance in software defect prediction [2], [8], [25]. Deep Belief Network (DBN) proposed by Wang [8] is also compared in our empirical studies. We compare the results of using ADTree, NB and LR on semantic features generated by the DBN model. For Cross-Project Defect Prediction (CPDP), we compare state-of-the-art CPDP method TCA+ in [15], which maps source projects and target projects onto the same subspace and generates new feature vectors based on transfer techniques. TCA+ is one of the best cross-project defect prediction methods.

We follow previous work [8] that using the data set from an older version project as a training set and predict the defective modules in the data set from the current version for evaluation. The parameters of CAP-CNN (Θ, W_c, W_m, U) are learned automatically during training process. We set the activation function in CNN as ReLU $\sigma(x) = \max(x, 0)$ and use filter windows (d) of 3,4,5 with 100 feature maps each. In addition, the numbers of comments feature to be eliminated during training each batch n_e is set according to the size of data. For example, suppose the size of data sets is n and the batch size is set as n_b, the eliminate number $n_e = n/n_b$, so that when all batches of training data fed into the model, the comments feature are all eliminated. We alose use the noise handling method from Kim's to prune the noisy data [26] and employ re-sampling method to make the training data sets more balance [12].

B. Experimental Results

We firstly investigate if the code comments features from source code help improve defect prediction performance. We compare the results of CAP-CNN and the original CNN model. The structure and parameters settings of CAP-CNN and CNN are exactly the same except that CAP-CNN uses code and comments as features while CNN only uses the source code for evaluation. For fair comparison, the testing module of both model does not contain comments part. We choose the newest version from each project for evaluation and set the previous version data as the training set. Fig. 5 shows the comparison results between CAP-CNN and CNN in terms of F-measure. It can be easily observed from Fig. 5 that CAP-CNN has a higher F-measure than CNN in 6 of 8 data sets. Apart from the difference by using comments or not during the training process, the model structure and parameters are exactly the same in CAP-CNN and CNN models, so it can be understood that the comments features help improve defect prediction performance.

The comparison results between CAP-CNN and state-of-the-art defect prediction for WPDP are detailed in Table. I. It can be clearly observed that the CAP-CNN model performs the best results among all compared methods. The CAP-CNN model performs the best F-measure values in 9 of 12 tasks. Comparing to deep feature extraction model DBN, CAP-CNN still improve the performance of DBN-ADTree (0.601) by11.34% , DBN-NB (0.568) by 17.78% and DBN-LR (0.568) by 20.18%. The reasons why CAP-CNN performs better may be summarized: 1. CAP-CNN generates semantic features under supervised learning, while DBN does not use label information for feature representation; 2. CAP-CNN combines source code and comments for software defect prediction, which generates a more semantic feature representation indicating the structure and functionality of source modules, while the DBN model only uses source code as features.

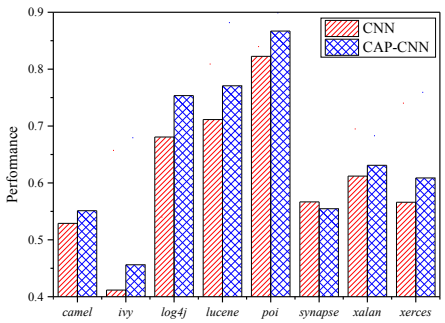


Figure 4. The comparison results (F-measure) between CAP-CNN and original CNN. CNN does not use code comments as features and CAP-CNN uses both code and comments features. It can be found that code comments help improves prediction results in most data sets.

Table I
COMPARISON RESULTS (F-MEASURE) OF CAP-CNN AND COMPARED METHODS FOR WITHIN-PROJECT DEFECT PREDICTION.

Project	Version	ADTree	NB	LR	DBN+ ADTree	DBN+ NB	DBN+ LR	CAP-CNN
camel	1.2⇒1.4	.373	.307	.363	**.785**	.459	.598	.623
	1.4⇒1.6	.391	.265	.346	.374	.481	.342	**.551**
ivy	1.4⇒2.0	.329	.389	.240	.350	.344	.348	**.456**
log4j	1.0⇒1.1	.687	.689	.535	.701	.725	.682	**.754**
lucene	2.0⇒2.2	.502	.500	.598	.651	.632	.630	**.743**
	2.2⇒2.4	.605	.378	.694	**.773**	.738	.629	.771
poi	1.5⇒2.5	.558	.323	.503	.640	.770	.664	**.891**
	2.5⇒3.0	.754	.462	.745	.803	.777	.783	**.867**
synapse	1.0⇒1.1	.476	.508	.316	.544	.479	.423	**.577**
	1.1⇒1.2	.533	.565	.533	**.583**	**.579**	.541	.555
xalan	2.4⇒2.5	.518	.398	.540	.595	.452	.565	**.631**
xerces	1.2⇒1.3	.238	.333	.266	.411	.380	.475	**.609**

We also investigate the comparison experiments for CPDP, and the results are shown in Table II. The first column indicates different Cross-Project Defect Prediction tasks. For example, the first row '*synapse-1.2* ⇒ *ivy-2.0*' indicates that using the data set from project '*synapse-1.2*' for training and predict the defective modules in project '*ivy-2.0*'. The results show that CAP-CNN achieves the best F-measure in 9 out of 13 pairs of CPDP tasks, while DBN-CP performs the best in 3 tasks and TCA+ gets the best performance on only one data set. The results indicate that the CAP-CNN model performs the best on most CPDP tasks. To compare the average F-measure of CPDP tasks, CAP-CNN is 61.6%, which is higher than the 55.3% of DBN-CP and the 46.3% of TCA+.

Table II
COMPARISON RESULTS (F-MEASURE) OF CAP-CNN AND COMPARED METHODS FOR CROSS-PROJECT DEFECT PREDICTION.

Task	DBN-CP	TCA+	CAP-CNN
synapse-1.2 ⇒ ivy-2.0	**.824**	.383	.727
xerces-1.3 ⇒ ivy-2.0	**.453**	.409	.429
log4j-1.1 ⇒ lucene-2.2	**.692**	.524	.685
xalan-2.5 ⇒ lucene-2.2	.594	.561	**.606**
synapose-1.2 ⇒ poi-3.0	.661	.343	**.731**
ivy-1.4 ⇒ synapse-1.1	.489	.348	**.574**
poi-2.5 ⇒ synapse-1.1	.425	.376	**.542**
ivy-2.0 ⇒ synapse-1.2	.433	**.570**	.538
poi-3.0 ⇒ synapse-1.2	.514	.542	**.667**
lucene-2.2 ⇒ xalan-2.5	.550	.530	**.696**
xerces-1.3 ⇒ xalan-2.5	.572	.581	**.677**
ivy-2.0 ⇒ xerces-1.3	.426	.394	**.524**
xalan-2.5 ⇒ xerces-1.3	.486	.398	**.588**
Average	.553	.463	**.616**

V. CONCLUSION

In this paper, we propose a novel software defect prediction model named CAP-CNN to automatically learn semantic features for software defect prediction by code comments embedding. To overcome the challenge that the testing modules do not contain comments, we propose a novel learning strategy of CAP-CNN to encode comments information in generating semantic features during training process. The results of empirical studies on widely-used data sets indicate that embedding code comments is able to generate a more representative semantic features and can improve software defect prediction performance. In the future, we would like to consider combining semi-supervised method to enrich the structure of deep learning model.

REFERENCES

[1] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Transactions on Software Engineering*, vol. 33, no. 1, pp. 2–13, 2007.

[2] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 485–496, 2008.

[3] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proceedings of the 31st International Conference on Software Engineering, Vancouver, Canada*, 2009, pp. 78–88.

[4] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener, "Defect prediction from static code features: current results, limitations, new approaches," *Automated Software Engineering*, vol. 17, no. 4, pp. 375–407, 2010.

[5] M. Li, H. Zhang, R. Wu, and Z.-H. Zhou, "Sample-based software defect prediction with active and semi-supervised learning," *Automated Software Engineering*, vol. 19, no. 2, pp. 201–230, 2012.

[6] S. Shivaji, E. J. Whitehead, R. Akella, and S. Kim, "Reducing features to improve code change-based bug prediction," *Transacations on Software Engineering*, vol. 39, no. 4, pp. 552–569, 2013.

[7] X.-Y. Jing, S. Ying, Z.-W. Zhang, S.-S. Wu, and J. Liu, "Dictionary learning based software defect prediction," in *Proceedings of the 36th International Conference on Software Engineering*, Hyderabad, India, 2014, pp. 414–423.

[8] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 38th International Conference on Software Engineering*, Austin, USA, 2016, pp. 297–308.

[9] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 531–540.

[10] T. Jiang, L. Tan, and S. Kim, "Personalized defect prediction," in *Proceedings of 28th International Conference on Automated Software Engineering, Silicon Valley, CA, USA*, 2013, pp. 279–289.

[11] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In, "Micro interaction metrics for defect prediction," in *Proceedings of the 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering and the 13rd European Software Engineering Conference, Szeged, Hungary*, Szeged, Hungary, 2011, pp. 311–321.

[12] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," in *Proceedings of the 37th International Conference on Software Engineering*, Florence, Italy, 2015, pp. 99–108.

[13] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proceedings of the 30th International Conference on Software Engineering*, Leipzig, Germany, 2008, pp. 181–190.

[14] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, A. Ihara, K.-i. Matsumoto, B. Ghotra, Y. Kamei, B. Adams, R. Morales, F. Khomh *et al.*, "The impact of mislabelling on the performance and interpretation of defect prediction models," in *Proceedings of the 37th International Conference on Software Engineering, Florence, Italy*, Florence, Italy, 2015, pp. 812–823.

[15] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *Proceedings of the 35th International Conference on Software Engineering*, San Francisco, CA, USA, 2013, pp. 382–391.

[16] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *Proceedings of IEEE International Conference on Software Qulity, Reliability and Security*, Lincoln, NE, USA, 2015, pp. 17–26.

[17] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proceedings of the 13th AAAI Conference on Artificial Intelligence*, Phoenix, AZ, USA, 2016, pp. 1287–1293.

[18] X. Huo, M. Li, and Z.-H. Zhou, "Learning unified features from natural and programming languages for locating buggy source code," in *Proceedings of the 25th International Joint Conference on Artificial Intelligence*, New York, NY, USA, 2016, pp. 1606–1612.

[19] H. Wei and M. Li, "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*, Melbourne, Australia, 2017, pp. 3034–3040.

[20] X. Huo and M. Li, "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*, Melbourne, Australia, 2017, pp. 1909–1915.

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

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

[23] M. Jureczko and L. Madeyski, "Towards identifying software project clusters with regard to defect prediction," in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering, Timisoara, Romania*, 2010, p. 9.

[24] F. Peters, T. Menzies, L. Gong, and H. Zhang, "Balancing privacy and utility in cross-company defect prediction," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1054–1068, 2013.

[25] E. J. Weyuker, T. J. Ostrand, and R. M. Bell, "Comparing the effectiveness of several modeling methods for fault prediction," *Empirical Software Engineering*, vol. 15, no. 3, pp. 277–295, 2010.

[26] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with noise in defect prediction," in *Proceedings of the 33rd International Conference on Software Engineering, Honolulu, HI, USA*, 2011, pp. 481–490.