2017 IEEE International Conference on Software Quality, Reliability and Security

Software Defect Prediction via Convolutional Neural Network

Jian Li*†, Pinjia He*†, Jieming Zhu*†, and Michael R. Lyu*†
*Department of Computer Science and Engineering, The Chinese University of Hong Kong, China
†Shenzhen Research Institute, The Chinese University of Hong Kong, China
{jianli, pjhe, jmzhu, lyu}@cse.cuhk.edu.hk

***Abstract*—To improve software reliability, software defect prediction is utilized to assist developers in finding potential bugs and allocating their testing efforts. Traditional defect prediction studies mainly focus on designing hand-crafted features, which are input into machine learning classifiers to identify defective code. However, these hand-crafted features often fail to capture the semantic and structural information of programs. Such information is important in modeling program functionality and can lead to more accurate defect prediction.**

In this paper, we propose a framework called Defect Prediction via Convolutional Neural Network (DP-CNN), which leverages deep learning for effective feature generation. Specifically, based on the programs' Abstract Syntax Trees (ASTs), we first extract token vectors, which are then encoded as numerical vectors via mapping and word embedding. We feed the numerical vectors into Convolutional Neural Network to automatically learn semantic and structural features of programs. After that, we combine the learned features with traditional hand-crafted features, for accurate software defect prediction. We evaluate our method on seven open source projects in terms of F-measure in defect prediction. The experimental results show that in average, DP-CNN improves the state-of-the-art method by 12%.

***Index Terms*—software reliability; software defect prediction; deep learning; CNN**

I. Introduction

With the ever-increasing scale of modern software, reliability has become a critical issue, since these software are often highly complicated and failure-prone. As the code defects[1] in the implementation of software are considered as the main causes of failures [1], to improve reliability, companies like Google employ code review and unit testing for finding bugs in fresh code [2]. However, manual code reviews are labor-intensive and testing all code units is impractical. As the software project budgets are finite, it would be beneficial to first check potentially buggy code. Therefore, software defect prediction techniques which automatically find potential bugs have been widely employed to help developers allocate their limited resources [3].

Software defect prediction [4], [5], [6], [7], [8] is a process of building classifiers to predict code areas that potentially contain defects, using information such as code complexity and change history. The prediction results (i.e., buggy code areas) can place warnings for code reviewers and allocate their efforts. The code areas could be files, changes or methods. In this paper, we focus on file-level defect prediction. Typical defect prediction is composed of two phases [9]: feature extraction from source files, and classifier development using various machine learning algorithms. Previous studies towards building more accurate predictions mainly focus on manually designing new discriminative features or new combinations of features, so that defects can be better distinguished. Traditional hand-crafted features include Halstead features based on the number of operators and operands [10], McCabe features based on dependencies [11], CK features for object-oriented programs [12], etc.

However, programs have well-defined *syntax* and rich *semantics* hidden in the Abstract Syntax Trees (ASTs), which traditional features often fail to capture. Thus the prediction results of traditional methods are not satisfactory enough. Recently, deep learning has emerged as a powerful technique for automated feature generation, since deep learning architecture can effectively capture highly complicated non-linear features. To make use of its powerful feature generation ability, the state-of-the-art method [13] leverages Deep Belief Network (DBN) in learning semantic features from token vectors extracted from programs' ASTs, which outperforms traditional features-based approaches in defect prediction. However, it overlooks the structural information of programs which can lead to more accurate defect prediction.

There are structural information in ASTs, specifying how adjacent tokens (i.e., nodes on ASTs) interact with each other to accomplish certain functionality. Slight difference in local structure may lead to huge variance in program results, even program crash. For example in Figure 1, there are two Java files, both of which contain a `for` statement, a `remove` function and an `add` function. The only difference between the two files is the order of the `remove` function and `add` function. $File2.java$ will encounter `NoSuchElementException`, when calls `remove` at the beginning if the queue is empty. Treating program as bag of words without order, the state-of-the-art methods often overlook this local structural information. As reported by deep learning researchers in speech recognition [14] and image classification [15], Convolutional Neural Network (CNN) is more advanced than DBN since the former can capture local patterns more effectively. Thus CNN is more capable of detecting local patterns such as the order difference in Figure 1, and conducting defect prediction.

[1]Defect and bug will be used interchangeably across this paper.

978-1-5386-0592-9/17 $31.00 © 2017 IEEE
DOI 10.1109/QRS.2017.42

```
1. static void myFunc (Queue myQueue) {
2.     int i;
3.     for (i = 0; i < 10; i++) {
4.         // insert i to the tail of the queue
5.         myQueue.add(i);
6.         myQueue.remove();
7.         // remove the head of the queue
8.     }
9. }
```

File1.java

```
1. static void myFunc (Queue myQueue) {
2.     int i;
3.     for (i = 0; i < 10; i++) {
4.         // remove the head of the queue
5.         myQueue.remove();
6.         myQueue.add(i);
7.         // insert i to the tail of the queue
8.     }
9. }
```

File2.java

Fig. 1. A motivating example. $File2.java$ will encounter an exception when calls *remove()* at the beginning if the queue is empty.

In addition, to better exploit the program context, word embedding technique [16] can be helpful as well. Word embedding maps each AST token into a numerical vector, which is trained regarding the context of each token. Consequently, tokens appearing in similar context tend to have similar vector representations that are close in the feature space, which can benefit CNN in learning the program semantics in certain contexts. Besides deep learning generated features, traditional defect prediction features such as complexity metrics and process metrics are shown to be informative in distinguishing buggy code [8], which may complement features generated by deep learning. Intuitively, by combining CNN and traditional features, we can get a richer feature representation of buggy source code.

In this paper, we propose a framework called Defect Prediction via Convolutional Neural Network (DP-CNN), which captures both semantic and structural features of programs. Specifically, we first parse source code into ASTs, and select representative nodes on ASTs to form token vectors. Thus each source file is represented by a token vector. Then we conduct mapping and word embedding, which converts the token vectors into numerical vectors, and input the numerical vectors to CNN. CNN will automatically generate semantic and structural features of the source code, which are then combined with several traditional defect prediction features. Finally the combined features are fed into a Logistic Regression classifier. We evaluate our method on seven open-source Java projects with well-established labels (i.e., buggy or clean), in terms of F-measure in defect prediction. The experimental results indicate that averagely, the proposed DP-CNN improves the state-of-the-art DBN-based method [13] by 12%, as well as traditional features-based method by 16%. In summary, This paper makes the following contributions:

- We propose a CNN-based defect prediction framework to automatically generate discriminative features from programs' ASTs, which preserves semantic and structural information of the source code.
- We employ word embedding to encode tokens extracted from ASTs, which benefits CNN in learning the semantics of programs.
- We combine the CNN-learned features with traditional defect prediction features, taking advantage of both non-linear features and hand-crafted features.

The rest of this paper is organized as follows. Section II introduces the background of defect prediction and CNN. Section III elaborates our proposed DP-CNN, which automatically learns semantic and structural features from source code for defect prediction. Section IV shows the experimental setup and results, including the parameter tuning. Section V and Section VI present the threats to validity and related work, respectively. Finally we conclude this paper and discuss plans for future work in Section VII.

II. Background

In this section, we briefly introduce the background of file-level defect prediction techniques and convolutional neural network. Here file-level means that each of the training or test instances is a source code file.

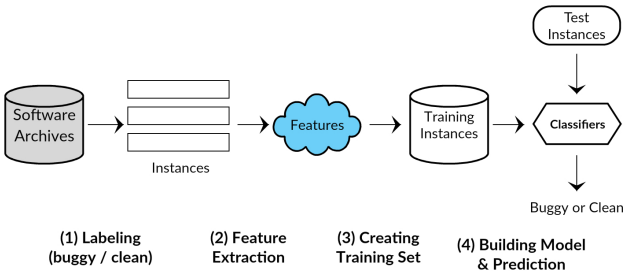


Fig. 2. Defect Prediction Process

A. Defect Prediction

Software defect prediction is a process of predicting code areas that potentially contain defects, which can help developers allocate their testing efforts by first checking potentially buggy code [3]. Defect prediction is essential to ensuring reliability of today's large-scale software. Figure 2 presents a typical file-level defect prediction process which is commonly adopted in the literature [5], [13], [17]. As the process shows, the first step is to collect source code files (instances) from software archives and label them as buggy or clean. The labeling process is based on the number of post-release defects of each file. A file is labeled as *buggy* if it contains at least one post-release bug. Otherwise, the file is labeled as *clean*. The second step is to extract features from each file. There are many traditional features defined in past studies, which can be categorized into two kinds: code metrics (e.g., McCabe features [11] and CK features [12]), and process metrics (e.g., change histories). The instances with the corresponding features and labels are subsequently employed to train predictive classifiers using various machine learning algorithms such as SVM, Naive Bayes, and Dictionary Learning [5]. Finally, new instances are fed into the trained classifier, which can predict whether the files are buggy or clean.

The set of instances used for building the classifier is *training set*, while *test set* includes the instances used for evaluating the learned classifier. In this work, we focus on within-project defect prediction, i.e., the training and test sets

belong to the same project. Following the previous work in this field [13], we use the instances from an older version of this project for training, and instances from a newer version for test.

B. Convolutional Neural Network

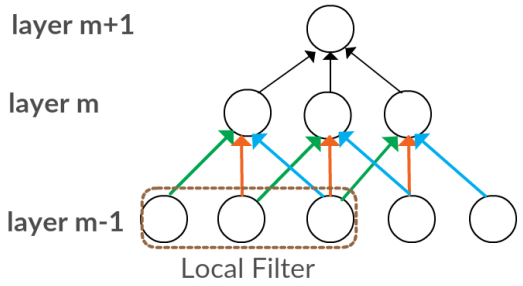


Fig. 3. A CNN architecture. The architecture depicts the two characteristics of CNN: Sparse Connectivity and Shared Weights, which enable CNN to capture local structural information of the inputs.

Convolutional Neural Networks (CNNs) are a specialized kind of neural networks for processing data that have a known grid-like topology [18], such as time-series data in 1D grid and image data in 2D grid. CNNs have been demonstrated successful in many practical fields, including speech recognition [14], image classification [15], [19] and natural language processing [20]. In this work, we leverage CNNs for effective feature generation from source code. Figure 3 depicts the general architecture of CNNs. Compared with traditional Artificial Neural Networks (ANNs) or Multi-Layer Perceptrons (MLPs), CNNs have two key characteristics: Sparse Connectivity and Shared Weights, which can benefit our defect prediction in capturing local structural information of programs.

Sparse Connectivity means CNNs employ a local connectivity pattern between neurons of adjacent layers to generate spatially local correlation of the inputs. For example in Figure 3, the inputs of units in hidden layer *m* are from a subset of units in layer *m-1*, which are spatially contiguous. The size of the subset is 3, so units in layer *m* only connect to 3 adjacent neurons in the layer below (i.e., neurons in the dashed rectangle), rather than connecting to all the neurons in traditional ANNs. Each subset acts as a local filter over the input vector, which can produce strong responses to a spatially local input pattern. Each local filter applies a non-linear transformation just like usual ANNs: multiplying the input with a linear filter, adding a bias term and then applying a non-linear function. In Figure 3, if we denote the i-th hidden unit in layer m as h_i^m, then the local filter in layer $m-1$ acts as follows (for *sigmoid* non-linearities):

$$h_i^m = sigmoid((W^{m-1} * x)_i + b^{m-1}). \quad (1)$$

where W^{m-1} and b^{m-1} denote the weights and bias of the local filter.

Shared Weights mean each filter shares the same parameterization (weight vector and bias). As our example in Figure 3, we show a local filter consisting of 3 units. Across the entire layer *m-1*, there are 3 local filters, and the same-colored arrows indicate they share the same weights. Replicating filters in this way enables us to detect features regardless of their position in the input vector. Moreover, weight sharing can greatly increase learning efficiency by reducing the number of free parameters.

Another important concept of CNNs is *max-pooling*, which partitions the output vector into several non-overlapping sub-regions, and outputs the maximum value of each sub-region. This is a smart way of reducing the dimensionality of intermediate representations and providing additional robustness to our defect prediction.

The effectiveness of CNNs largely depends on the parameters, such as filter length and batch size. The model would not even converge under a bad parameter setting. Thus parameter tuning is a key to train a successful CNN. We will discuss how to set these parameters in Section IV-F.

III. Approach

In this section, we elaborate our proposed DP-CNN, a framework which automatically generates semantic and structural features from source code and combines traditional features, for accurate software defect prediction. Figure 4 illustrates the overall workflow of DP-CNN.

As the workflow shows, we first parse source code of both training files and test files into Abstract Syntax Trees (ASTs), then select representative nodes on ASTs to form token vectors. Thus each source file becomes a token vector, which is fed into the following encoding phase. We build a mapping between integers and tokens, and employ word embedding to encode token vectors as numerical vectors which are input to subsequent CNN. CNN automatically generates semantic and structural features of source code from the input vectors, which are then combined with several traditional defect prediction features. This feature generation process is elaborated in Figure 6. Finally, the combined features are fed into a Logistic Regression classifier. After building our classifier model (i.e., deciding the weights and biases in CNN and Logistic Regression), we can produce a probability for each fresh code file, indicating whether it is buggy or clean.

To sum up, our approach has of four major steps: 1) parsing source code into ASTs and extract tokens, 2) encoding token vectors into numerical vectors, 3) employing CNN to generate semantic and structural features and combining traditional defect prediction features, and 4) building a Logistic Regression classifier to decide whether the fresh code files are buggy or clean.

A. Parsing Source Code

In order to represent each source code file as a vector, we should first answer a fundamental question: what is the proper granularity of representation? In general, vector representations map a symbol to a real-valued, distributed vector. For software programs, possible granularities of the symbol include character-level, token-level, nodes on ASTs, etc. As analyzed in [21], only nodes on ASTs are a suitable granularity

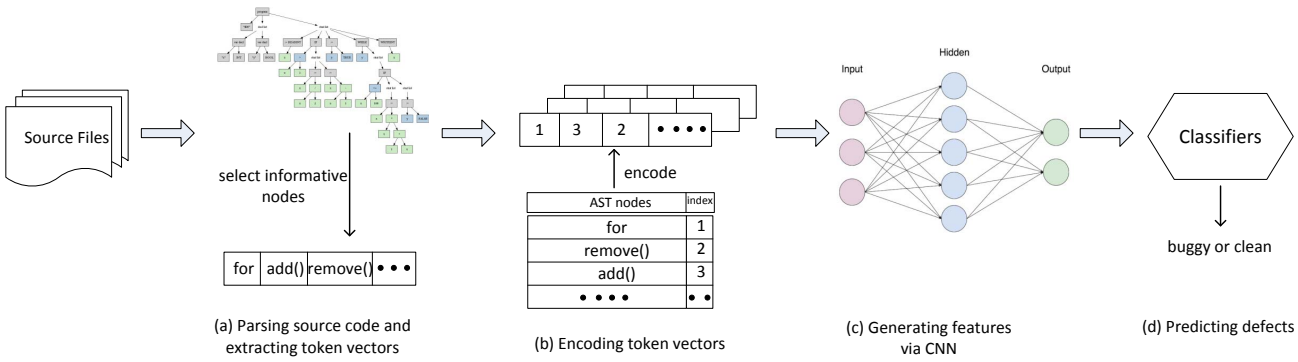


Fig. 4. The overall workflow of our proposed DP-CNN.

to build program representation, which preserve both syntactic and structural information of the programs. In our experiments, we employ an open-source Python package called javalang[2] to parse our Java source code into ASTs. Javalang provides a lexer and parser targeting Java 8, whose implementation is based on the Java language specification.

Following the state-of-the-art method [13], we only select three types of nodes on ASTs as tokens: 1) nodes of method invocations and class instance creations, which are recorded as their method names or class names, 2) declaration nodes, i.e., method declarations, type declarations, and enum declarations, whose values are extracted as our tokens, and 3) control-flow nodes including `IfStatement`, `WhileStatement`, `ForStatement`, `ThrowStatement`, `CatchClause`, etc. Control-flow nodes are simply recorded as their node types. We exclude other types of AST nodes such as `Assignment` because they are often method-specific or class-specific, which do not have consistent meanings throughout the whole project. All the selected AST nodes are listed in Figure 5.

In this way, we convert each source file into a token vector. Take the two Java files in Figure 1 as example. After the tokenization as described, $File1.java$ and $File2.java$ will be denoted as `[<FOR>, add, remove]` and `[<FOR>, remove, add]`, respectively.

B. Encoding Tokens and Handling Imbalance

1) Encoding Tokens: Since CNNs require inputs as numerical vectors, the extracted token vectors cannot be directly sent to a CNN. To solve this problem, we first build a mapping between integers and tokens, and convert token vectors into integer vectors. Each token is associated with a unique integer identifier which ranges from 1 to the total number of token types. In this way, the same tokens keep as the same identifier and different tokens such as different method names and class names still remain different. Also, CNNs require input vectors to have the same length. But our converted integer vectors may differ in their lengths. In response, we simply append 0 to each integer vectors, making their lengths consistent with the longest vector. 0 does not have any meaning since we encode tokens starting from 1. Additionally, during the encoding process, we filter out infrequent tokens which might be designed for a specific file and not generalized for other files. Specifically, we only encode tokens occurring three or more times, while denote the others as zeros.

[2]https://github.com/c2nes/javalang

FormalParameter	*ForStatement*
BasicType	*AssertStatement*
PackageDeclaration	*BreakStatement*
InterfaceDeclaration	*ContinueStatement*
CatchClauseParameter	*ReturnStatement*
ClassDeclaration	*ThrowStatement*
MethodInvocation	*SynchronizedStatement*
SuperMethodInvocation	*TryStatement*
MemberReference	*SwitchStatement*
SuperMemberReference	*BlockStatement*
ConstructorDeclaration	*StatementExpression*
ReferenceType	*TryResource*
MethodDeclaration	*CatchClause*
VariableDeclarator	*CatchClauseParameter*
IfStatement	*SwitchStatementCase*
WhileStatement	*ForControl*
DoStatement	*EnhancedForControl*

Fig. 5. The selected AST nodes

As discussed in Section I, we also employ word embedding [16] in the encoding phase. However, our word embedding is built and trained at the same time as CNN. So we wrap word embedding as a part of our CNN architecture and discuss it in the following CNN part.

2) Handling Imbalance: Software defect data are often imbalanced, in which the number of buggy instances is much less than the number of clean instances. Imbalanced data will degrade the performance of our model. To address this problem, two approaches are feasible. One approach is to reduce the training instances from the majority class (i.e., the clean files), while another approach is to duplicate training instances from the minority class (i.e., the buggy files). As the first approach would lose some information, we use the

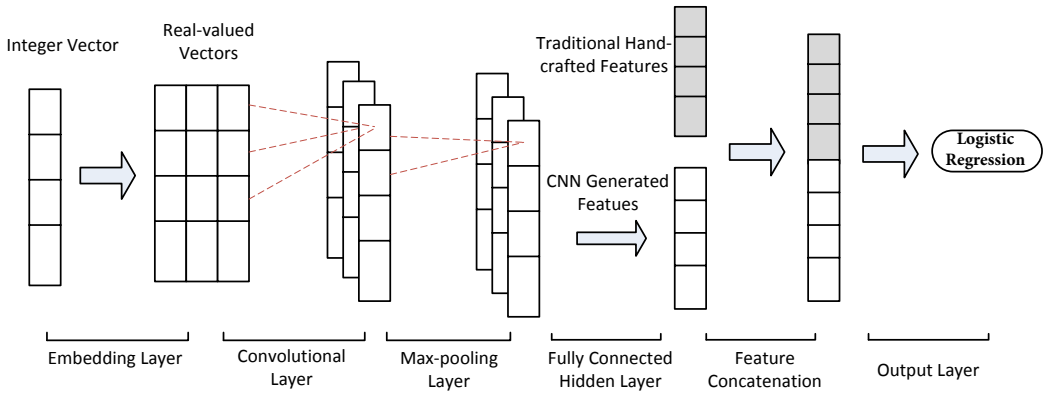


Fig. 6. The feature generation process of DP-CNN, which elaborates the step (c) in Figure 4.

second approach and duplicate the buggy files several times until we have a balanced dataset. Note that we only apply this process on the training files.

C. Building CNN and Combining Traditional Features

1) Building CNN: As discussed in Section II-B, we take advantage of CNN's powerful ability of feature generation, and capture semantic and local structural information of the source code. We train our CNN (i.e., the weights and biases in CNN) by using the training data. Considering that this work engages CNN only as an application, we adopt a standard architecture of CNN rather than fancy and complex architectures in some theoretical approaches [15], [19]. In particular, our CNN consists of an embedding layer (i.e., word embedding), a convolutional layer, a max-pooling layer, a fully-connected hidden layer, and finally a single unit output layer working as a Logistic Regression classifier (the last step in Figure 4). The overall architecture is illustrated in Figure 6. Except the output layer uses *sigmoid* activation function, all other layers use *ReLU* activation function. Our implementation is based on *Keras*[3], through which we can easily and quickly build neural networks. There is an example in *Keras* demonstrating the use of 1D Convolutional Neural Network for text classification, which is taken as our reference.

As shown in Figure 6, we employ word embedding [16] as the first layer, which turns positive integers (indexes) into real-valued vectors of fixed size. Obviously, a simple index does not carry much context information about the token extracted from ASTs. However, word embedding is trained regarding the context of each token. A feature vector will be learned for each token, and tokens appearing in similar context tend to have similar vector representations which have close distance in the feature space. In this way, we can further exploit the semantics of programs via CNN. A word embedding is defined as $f : M \rightarrow \mathbb{R}^n$, where M represents the dictionary of words (or tokens in this work), f is a parameterized function mapping words to n-dimensional vectors. The parameters of word embedding are initialized randomly and learned at the same time as other parameters in the following CNN architecture. The implementation of word embedding is also based on *Keras*. Therefore, we wrap word embedding as a part of our CNN architecture and discuss them together in this paper. Hereinafter, the term *CNN* always includes word embedding.

Our CNN model is trained using the minibatch stochastic gradient descent (SGD) algorithm [22], with the Adam optimizer [23]. We will discuss the details of parameter tuning such as batch sizes and the number of training epochs in Section IV-F.

2) Combining Traditional Features: Till now we only consider static code features through the CNN. However, in conventional defect prediction methods, other features such as complexity metrics and process metrics are shown to be informative in distinguishing buggy code [8]. In fact, in our dataset we are provided with several traditional defect prediction features of each file, which are carefully extracted by the dataset contributors. To make use of these information, we directly concatenate the CNN-learned feature vectors with traditional hand-crafted feature vectors. This concatenation can be realized via `Merge` operator in *Keras*. Finally, the combined feature vectors are fed into the subsequent Logistic Regression classifier. To demonstrate the effectiveness of combining traditional features, we design a variant of DP-CNN which directly feeds the CNN-learned features to final classifier without concatenation. In the experiments part, we will compare this variant with DP-CNN, as well as other state-of-the-art methods.

D. Predicting Defects

We employ Logistic Regression as the final classifier, since it is widely used in the literature [24] and we mainly focus on feature generation in this paper. We process each file in both training set and test set following the above steps, and obtain semantic and structural features of each source file. After we train our model using the training files with their corresponding labels, both the weights and the biases in our CNN and Logistic Regression are fixed. Then for each file in

[3]Keras (http://keras.io) is a popular Deep Learning library for Python.

TABLE I
DATASET DESCRIPTION

Project	Description	Versions (Tr, T)	Avg. Files	Buggy Rate (%)
camel	Enterprise integration framework	1.4, 1.6	892	18.6
jEdit	Text editor designed for programmers	4.0, 4.1	284	23.8
lucene	Text search engine library	2.0, 2.2	210	55.7
xalan	A library for transforming XML files	2.5, 2.6	815	48.5
xerces	XML parser	1.2, 1.3	441	15.5
synapse	Data transport adapters	1.1, 1.2	239	30.5
poi	Java library to access Microsoft format files	2.5, 3.0	409	64.7

the test set, we feed it into our defect prediction model and the final classifier will give us a value, indicating the probability of this file being *buggy*.

IV. EVALUATION

In this section, we evaluate the effectiveness of our DP-CNN by comparing its accuracy on defect prediction with other state-of-the-art methods. In particular, our evaluation addresses the following research questions (RQ):

- *RQ1: Do the deep learning-based methods outperform traditional features-based methods?*
- *RQ2: Does DP-CNN which combines traditional features outperform deep learning-based methods?*
- *RQ3: How is the performance of DP-CNN under different parameter settings?*

All our experiments were run on a Linux server with one Tesla K40m GPU. Unless otherwise stated, each experiment was run for ten times and the average results were reported.

A. Evaluation Metrics

To evaluate the prediction accuracy, we use a widely adopted metric in the literature [8], [17]: the *F-measure* (also *F1 score*), which is the harmonic mean of *precision* and *recall* [25].

We first present some notations in defining precision, recall, and F-measure: (i) predicting a buggy file as buggy ($b \to b$); (ii) predicting a buggy file as clean ($b \to c$); and (iii) predicting a clean file as buggy ($c \to b$). N denotes the number of files in each above situation, e.g., $N_{b\to b}$ for the first case. Then our metrics can be defined as follows:

Precision: The ratio of the number of files correctly classified as buggy to the number of files classified as buggy.

$$\text{Precision: } P = \frac{N_{b\to b}}{N_{b\to b} + N_{c\to b}} \tag{2}$$

Recall: The ratio of the number of files correctly classified as buggy to the number of truly buggy files.

$$\text{Recall: } R = \frac{N_{b\to b}}{N_{b\to b} + N_{b\to c}} \tag{3}$$

F-measure: The traditional F-measure (F1 score) is the harmonic mean of precision P and recall R.

$$\text{F-measure: } F = \frac{2 * P * R}{P + R} \tag{4}$$

Usually, there are trade-offs between precision and recall. For example, by predicting all the test files as buggy, we will get a high recall as 1 but a very low precision. Therefore, F-measure is a composite measure of precision and recall which falls in the range [0, 1]. The higher the F-measure is, the better the prediction performance represents.

B. Dataset Description

Our defect prediction dataset comes from tera-PROMISE Repository[4], which is a publicly available repository specializing in software engineering research datasets. We select seven open-sourced Java projects from this repository, where the version numbers, the class name of each file, and most importantly, the defect label for each source file are provided. With the version numbers and class names, we can extract source code of each file from Github[5] and apply it in our DP-CNN framework. Table I shows the details of these projects, including project description, versions, the average number of files, and the average buggy rate. To obtain the training and test data, following the state-of-the-art method [13], we use files from two consecutive versions of each project. The older version is denoted as *Tr* and the newer version is denoted as *T*. In average, the number of files in each project is 330 and the buggy rate of each project is 35%.

Moreover, in this dataset, we are provided with 20 traditional defect prediction features for each source file, including Lines of Code (LOC), Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), and McCabe complexity measures (Max_CC and Avg_CC), etc. The 20 traditional features are carefully extracted by Jureczko et al., the dataset contributors [26]. We list the detailed description about the 20 features in Table III. These features and data have been widely used in previous studies [3], [5], [17].

C. Baseline Methods

We compare our proposed **DP-CNN** for defect prediction with the following baseline methods:

- **DBN [13]:** the state-of-the-art method which employs Deep Belief Network (DBN) on source code to extract semantic features for defect prediction.

[4]http://openscience.us/repo/defect/
[5]https://github.com/apache

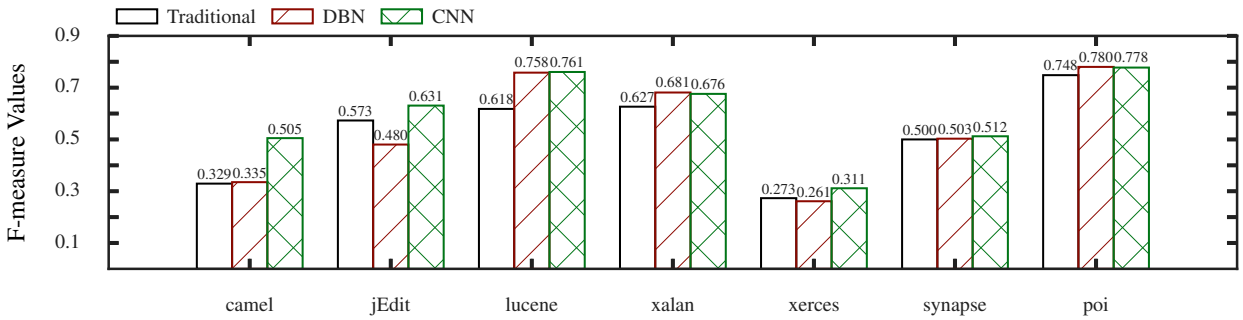


Fig. 7. Performance comparison between traditional method and deep learning-based methods for defect prediction.

- **Traditional [24]:** traditional method which builds a Logistic Regression classifier based on the 20 features.
- **DBN+:** an improved version of DBN proposed by us, which combines the semantic features with the traditional features as we do in this work.
- **CNN:** a variant of DP-CNN which directly feeds the CNN-learned features to final classifier without combining traditional features.

When implementing DBN and DBN+, we use the same network architecture and parameter settings as in [13], i.e., 10 hidden layers and 100 nodes in each hidden layer. For a fair comparison, we follow the same process and tools in our work to parse source code and encode tokens, as well as handling data imbalance.

D. Performance of Deep Learning-based Methods (RQ1)

We first compare the traditional features-based method for defect prediction with two deep learning-based methods, i.e., DBN and CNN, as introduced in the previous section. The purpose of this comparison is to validate the advantage of deep learning techniques in software defect prediction. We conduct seven sets of defect prediction experiments on those projects listed in Table I, within each of which the older version is used to train prediction models, and the newer version is used as the test set to evaluate the trained models.

Figure 7 shows the experimental results, i.e., F-measure values on each project by applying the three competing methods. We take project *camel* as an example. After training using version 1.4 and testing using version 1.6, the F-measure of defect prediction is 0.329, 0.335, and 0.505 for traditional method, DBN and CNN, respectively. Both DBN and CNN outperforms traditional method. We can see from figure 7 that, in most cases, traditional method achieves the lowest F-measure, indicating the advantages of deep learning-based methods over the traditional ones. More significantly, CNN achieves the best performance. These results not only validate the effectiveness of the state-of-the-art DBN method [13], but also justify our proposed CNN method, which performs even better than DBN.

TABLE II
PERFORMANCE COMPARISON OF DIFFERENT MODELS

Project	Traditional	DBN	DBN+	CNN	DP-CNN
camel	0.329	0.335	0.375	0.505	**0.508**
jEdit	0.573	0.480	0.549	**0.631**	0.580
lucene	0.618	0.758	0.761	0.761	**0.761**
xalan	0.627	0.681	0.681	0.676	**0.696**
xerces	0.273	0.261	0.276	0.311	**0.374**
synapse	0.500	0.503	0.486	0.512	**0.556**
poi	0.748	0.780	0.782	0.778	**0.784**
Average	0.524	0.543	0.559	0.596	**0.608**

E. Performance of Combining Traditional Features (RQ2)

After validating the effectiveness of deep learning-based methods in defect prediction, we continue to improve DBN and CNN via combining traditional features as described in Section III-C. Taking both traditional hand-crafted features and deep-learning based features into consideration, we can expect to achieve more accurate prediction models. We consequently run experiments on the seven projects with all the five models, including our proposed DP-CNN. As before, the older version of each project is used as the training set, while the newer version is used as the test set. The corresponding F-measure values of all models are listed in Table II.

In Table II, each row represents each project, with the project name in first column and F-measure of each method in the rest five columns. Note that the highest value of each row is marked in bold. The results generally validate our intuition that including the traditional features into the deep learning-based methods can improve the prediction accuracy. For example, when applying different methods on project *camel*, DBN+ produces an F-measure of 0.375, which is 4% higher than DBN, while DP-CNN achieves the highest F-measure as 0.508, which is a bit better than CNN's 0.505. Among all the experiments, DP-CNN achieves the best performance in six out of the seven projects. The only exception is *jEdit*, where DP-CNN produces a lower F-measure than that of CNN. The reason may be that the training set of *jEdit* is relatively

small, thus adding traditional features to CNN may cause overfitting and degrade the performance. Also, in most cases DBN+ performs slightly better than DBN.

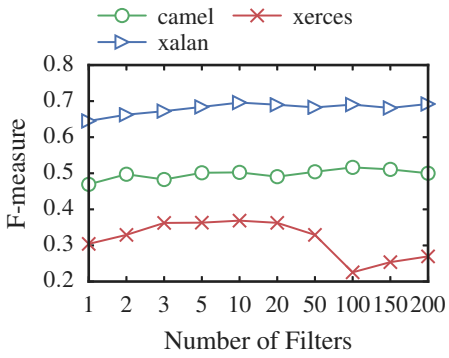
As shown in last row of Table II, in average, the order of model accuracy from the lowest to the highest is Traditional, DBN, DBN+, CNN, and DP-CNN. More specifically, DP-CNN improves Traditional, DBN and CNN by 16%, 12% and 2%, respectively. This exactly answers our RQ2 that combining traditional features is beneficial, and DP-CNN performs the best.

F. Performance under Different Parameter Settings (RQ3)

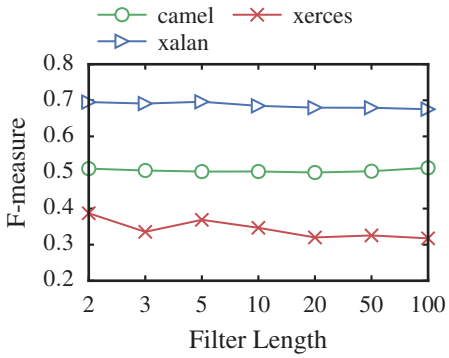
In this section, we discuss how we set the free parameters in DP-CNN for achievement of the best performance. Due to space limitations, here we only analyze three parameters which are key to CNNs: the number of filters, the filter length, and the number of nodes in hidden layers. We vary the values of these three parameters and conduct experiments on project *camel*, *xalan*, and *xerces* respectively. For other parameters, we directly present their values which are obtained via our validation: batch size is set as 32, the training epoch is 15, and the embedding dimension is set as 30.

Figure 8 shows the F-measure of DP-CNN under different number of filters, different filter length and different number of hidden nodes. We can see that the optimal number of filters is 10, where the three curves generally reach the peak, while the filter length makes little difference in F-measure. Considering that the larger the filter length is, the more running time we will take, we choose the filter length as 5. The number of hidden nodes is set as 100, which is similar to the number of filters that the three curves roughly peak at 100.

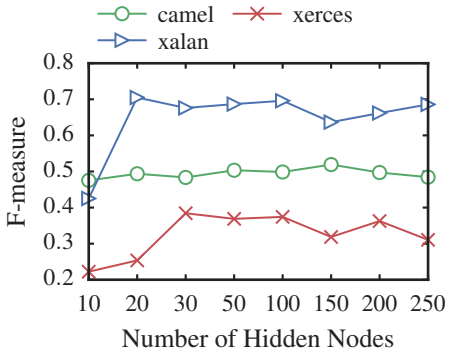
Obviously these optimal model parameters are application dependent. But with our automated DP-CNN framework, they are not difficult to determine when the relevant data are available.



(a) Different number of filters



(b) Different filter length



(c) Different number of hidden nodes

Fig. 8. Performance of DP-CNN under different parameter settings.

V. Threats to Validity

A. Implementation of DBN

To evaluate the performance our method in defect prediction, we compare our proposed DP-CNN with DBN [13], which is the state-of-the-art defect prediction technique. Since the original implementation of DBN is not released, we have reimplemented our own version of DBN via *Keras*, as well as DBN+. Although we strictly followed the procedures and parameters settings described in the DBN paper, some implementation details were still not mentioned in the paper, such as the full list of selected AST nodes and the learning rate when training neural networks. Thus our new implementation may not reflect all the implementation details of the original DBN method. However, we have consulted the first author of the DBN paper via email regarding key implementation details, and we are confident that our implementation is very close to the original DBN.

B. Datasets Selection

We conducted our experiments using seven open-source projects in the PROMISE data set, they might not be representative of all software projects. Besides, we only evaluated DP-CNN on projects written in Java language. Given projects that are not included in the seven projects or written in other programming languages (e.g., C++ or Python), our proposed method might generate better or worse results. To make DP-CNN more generalizable, in the future, we will conduct

TABLE III
DESCRIPTION OF THE 20 TRADITIONAL FEATURES IN PROMISE DATASETS [24]

Feature Names	Symbols	Description
Weighted Methods per Class	WMC	The number of methods in the class.
Depth of Inheritance Tree	DIT	Indicates the position of the class in the inheritance tree.
Number of Children	NOC	The number of immediate descendants of the class.
Coupling Between Object classes	CBO	The value increases when the methods of one class access services of another.
Response for a Class	RFC	Number of methods invoked in response to a message to the object.
Lack of Cohesion in Methods	LCOM	Number of pairs of methods that do not share a reference to an instance variable.
Lack of Cohesion in Methods, different from LCOM	LCOM3	If m, a are the number of methods, attributes in a class number and $\mu(a)$ is the number of methods accessing an attribute, then $\text{lcom3} = \left(\left(\frac{1}{a}\sum_j{}^{a}\mu(a_j)\right) - \text{m}\right)/(1-\text{m})$.
Number of Public Methods	NPM	The number of all the methods in a class that are declared as public.
Data Access Metric	DAM	Ratio of the number of private (protected) attributes to the total number of attributes.
Measure of Aggregation	MOA	The number of data declarations (class fields) whose types are user defined classes.
Measure of Function Abstraction	MFA	Number of methods inherited by a class plus number of methods accessible by member methods of the class.
Cohesion among Methods of class	CAM	Summation of number of different types of method parameters in every method divided by a multiplication of number of different method parameter types in whole class and number of methods.
Inheritance Coupling	IC	The number of parent classes to which a given class is coupled.
Coupling Between Methods	CBM	Total number of new/redefined methods to which all the inherited methods are coupled.
Average Method Complexity	AMC	The number of JAVA byte codes.
Afferent couplings	Ca	How many other classes use the specific class.
Efferent couplings	Ce	How many other classes is used by the specific class.
Maximum McCabe	Max (CC)	Maximum McCabe's Cyclomatic Complexity values of methods in the same class.
Average McCabe	Avg (CC)	Average McCabe's Cyclomatic Complexity values of methods in the same class.
Lines of Code	LOC	Measures the volume of code.

experiments on a variety of projects including open-source and closed-source projects, and extend our method to other programming languages.

VI. RELATED WORK

A. Software Defect Prediction

Software defect prediction is an active research area in *Software Engineering* [3], [4], [5], [6], [7], [13], [27], [28]. In the literature, most defect prediction techniques focus on manually designing new discriminative features or new combinations of features from labeled historical defect data, which are fed into machine learning based classifiers to identify code defects [3]. Commonly used features can be categorized into static code features and process features [8]. Static code features [11], [12], [29] were already introduced in previous sections, here we discuss the process features which were proposed recently and used in defect prediction. Moser et al. [4] employed the number of revisions, authors, past fixes, and ages of files as features to predict defects. Nagappan et al. [30] proposed code churn features, and shown that these features were effective for defect prediction. Moreover, Hassan et al. [31] used entropy of change features in predicting defect. Based on these code and process features, many machine learning models are built for two different defect prediction tasks: within-project defect prediction and cross-project defect prediction.

Within-project defect prediction means that both the training data and test data come from the same project, just like our work in this paper. When people proposed defect prediction for the first time in 1971 [32], they meant within-project. This problem is well explored in the past. For example, Elish et al. [33] evaluated the feasibility of Support Vector Machine (SVM) in predicting defect-prone software modules, and they compared SVM with other eight statistical learning methods on four NASA datasets. Amasaki et al. [34] proposed to employ the Bayesian Belief Network in predicting the final quality of a software product. Wang et al. [35] examined C4.5 in defect prediction, which is a kind of Decision Tree (DT) algorithm. Their results indicated that tree-based algorithms could generate good predictions. Moreover, Jing et al. [5] introduced the dictionary learning techniques to defect prediction. Their cost-sensitive dictionary learning based approach could significantly improve defect prediction in their experiments.

Recently, more and more papers studied the cross-project defect prediction problem, where the training data and test data come from different projects. Zimmermann et al. [36] evaluated the performance of cross-project defect prediction on 12 projects and their 622 combinations. They found the defect prediction models at that time could not adapt well to cross-project defect prediction. Premraj et al. [37] compared network and code metrics for defect prediction, and further built six cross-project defect prediction models using those metrics sets. Their results confirmed that cross-project defect prediction is a challenging problem. The state-of-the-art cross-project defect prediction is proposed by Nam et al. [7], who adopted a state-of-the-art transfer learning technique called Transfer Component Analysis (TCA). They further improved TCA as TCA+ by optimizing TCA's normalization process. They evaluated TCA+ on eight open-source projects, and the results shown their approach significantly improved cross-project defect prediction.

Our proposed DP-CNN differs from aforementioned defect prediction approaches in that, we utilize deep learning technique (i.e., CNN) to automatically generate discriminative

features from source code, rather than manually designing features, which can capture semantic and structural information of programs and lead to more accurate prediction.

B. Deep Learning in Software Engineering

Hindle et al. [38] were the first to demonstrate that software corpora has a "naturalness" property, such that real programs (or source code) written by real people have rich statistical properties. They successfully applied n-grams which is a statistical language model on software languages, to accomplish a code completion task. Recently, deep learning techniques have been adopted in Software Engineering to improve information retrieval-based tasks, due to its powerful ability for feature generation and remarkable achievements in other fields [14], [15], [20]. Specifically, Lam et al. [39] proposed to combine deep learning with information retrieval to improve the performance in localizing buggy files for bug reports. Huo et al. [40] employed CNN on source code in programs and natural language in bug reports to learn features respectively, and then combined the two kinds of features as unified features for bug localization. Raychev et al. [41] employed Recurrent Neural Network (RNN) to tackle code completion task, which was further improved by their later Decision Trees-based method [42]. White et al. [43] proposed to model sequential software languages using deep learning and applied their models in the task of code clone detection. Mou et al. [44] proposed tree-based CNN to better model source code while preserving the structural information, which was employed to classify the functionalities of programs. Gu et al. [45] utilized the RNN encoder-decoder model to address the problem of retrieving API call sequences, based on the user's natural language queries. Besides, Program Synthesis [46], [47], [48], [49] also becomes an active research area based on deep learning techniques.

Deep learning is also applied in defect prediction. Yang et al. [50] applied DBN on 14 existing change level features to generate new feature, for change level defect prediction. Wang et al. [13] further applied DBN on token vectors which are extracted from programs' ASTs, for file level defect prediction. Our work differs from the first work in that, we leverage deep learning to generate features directly from source code, rather than existing features. Our work differs from the second work in that, we employ CNN to capture the structural information of programs, and utilize word embedding and combining traditional features to further improve our prediction results.

VII. Conclusion and Future Work

With the ever-increasing scale and complexity of modern software, reliability assurance has become a significant challenge. To enhance the reliability of software, in this paper, we focus on predicting potential code defects in the implementation of software, thus reduce the workload of software maintenance. Specifically, we propose a defect prediction framework called DP-CNN (Defect Prediction via Convolutional Neural Network), which utilizes CNN for automated feature generation from source code with the semantic and structural information preserved. Besides, we employ word embedding and combine the CNN-learned features with traditional hand-crafted features, to further improve our defect prediction. Our experiments on seven open source projects show that averagely, DP-CNN improves the state-of-the-art DBN-based and traditional features-based methods by 12% and 16% respectively, in terms of F-measure in defect prediction.

To make DP-CNN more generalizable, in the future, we will conduct experiments on more projects, and extend our method to other programming languages like Python. Moreover, the results in this work demonstrate the feasibility of deep learning techniques in the filed of program analysis. It is promising to adapt deep learning in other software engineering tasks such as code completion and code clone detection, which will be our future work.

Acknowledgment

The work described in this paper was supported by the National Natural Science Foundation of China (Project No. 61472338), the Research Grants Council of the Hong Kong Special Administrative Region, China (No. CUHK 14234416 of the General Research Fund), and 2015 Microsoft Research Asia Collaborative Research Program (Project No. FY16-RES-THEME-005).

References

[1] A. G. Liu, E. Musial, and M.-H. Chen, "Progressive reliability forecasting of service-oriented software," in *ICWS'11: Proc. of the International Conference on Web Services*, 2011.

[2] Bug prediction at google. [Online]. Available: http://google-engtools.blogspot.hk/

[3] 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.

[4] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *ICSE'08: Proc. of the International Conference on Software Engineering*, 2008.

[5] X.-Y. Jing, S. Ying, Z.-W. Zhang, S.-S. Wu, and J. Liu, "Dictionary learning based software defect prediction," in *ICSE'14: Proc. of the International Conference on Software Engineering*, 2014.

[6] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," in *ICSE'15: Proc. of the International Conference on Software Engineering-Volume 2*, 2015.

[7] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *ICSE'13: Proc. of the International Conference on Software Engineering*, 2013.

[8] J. Nam, "Survey on software defect prediction," *Department of Compter Science and Engineerning, The Hong Kong University of Science and Technology, Tech. Rep*, 2014.

[9] M. R. Lyu *et al.*, *Handbook of software reliability engineering*. IEEE computer society press CA, 1996, vol. 222.

[10] H. H. Maurice, "Elements of software science (operating and programming systems series)," 1977.

[11] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, no. 4, pp. 308–320, 1976.

[12] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, 1994.

[13] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *ICSE'16: Proc. of the International Conference on Software Engineering*, 2016.

[14] O. Abdel-Hamid, A.-r. Mohamed, H. Jiang, and G. Penn, "Applying convolutional neural networks concepts to hybrid nn-hmm model for speech recognition," in *ICASSP'12: Proc. of the International Conference on Acoustics, Speech and Signal Processing*, 2012.

[15] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *NIPS'12: Proc. of the Advances in Neural Information Processing Systems*, 2012.

[16] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *NIPS'13: Proc. of the Advances in Neural Information Processing Systems*, 2013.

[17] 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.

[18] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[19] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, 1998.

[20] X. Zhang, J. Zhao, and Y. LeCun, "Character-level convolutional networks for text classification," in *NIPS'15: Proc. of the Advances in Neural Information Processing Systems*, 2015.

[21] L. Mou, G. Li, Y. Liu, H. Peng, Z. Jin, Y. Xu, and L. Zhang, "Building program vector representations for deep learning," *arXiv preprint arXiv:1409.3358*, 2014.

[22] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*, 2010.

[23] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[24] Z. He, F. Peters, T. Menzies, and Y. Yang, "Learning from open-source projects: An empirical study on defect prediction," in *ESEM'13: Proc. of the International Symposium on Empirical Software Engineering and Measurement*, 2013.

[25] C. D. Manning, P. Raghavan, H. Schütze *et al.*, *Introduction to information retrieval*. Cambridge University Press, 2008, vol. 1, no. 1.

[26] M. Jureczko and L. Madeyski, "Towards identifying software project clusters with regard to defect prediction," in *Proc. of the International Conference on Predictive Models in Software Engineering*, 2010.

[27] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller, "Predicting faults from cached history," in *ICSE'07: Proc. of the International Conference on Software Engineering*, 2007.

[28] S. Zhang, J. Ai, and X. Li, "Correlation between the distribution of software bugs and network motifs," in *QRS'16: Proc. of the International Conference on Software Quality, Reliability and Security*, 2016.

[29] R. Harrison, S. J. Counsell, and R. V. Nithi, "An evaluation of the mood set of object-oriented software metrics," *IEEE Transactions on Software Engineering*, vol. 24, no. 6, pp. 491–496, 1998.

[30] N. Nagappan and T. Ball, "Using software dependencies and churn metrics to predict field failures: An empirical case study," in *ESEM'07: Proc. of the International Symposium on Empirical Software Engineering and Measurement*, 2007.

[31] A. E. Hassan, "Predicting faults using the complexity of code changes," in *ICSE'09: Proc. of the International Conference on Software Engineering*, 2009.

[32] F. Akiyama, "An example of software system debugging." in *IFIP Congress (1)*, vol. 71, 1971, pp. 353–359.

[33] K. O. Elish and M. O. Elish, "Predicting defect-prone software modules using support vector machines," *Journal of Systems and Software*, vol. 81, no. 5, pp. 649–660, 2008.

[34] S. Amasaki, Y. Takagi, O. Mizuno, and T. Kikuno, "A bayesian belief network for assessing the likelihood of fault content," in *ISSRE'03: Proc. of the International Symposium on Software Reliability Engineering*, 2003.

[35] J. Wang, B. Shen, and Y. Chen, "Compressed c4. 5 models for software defect prediction," in *QSIC'12: Proc. of the International Conference on Quality Software*, 2012.

[36] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: a large scale experiment on data vs. domain vs. process," in *FSE'09: Proc. of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The foundations of Software Engineering*, 2009.

[37] R. Premraj and K. Herzig, "Network versus code metrics to predict defects: A replication study," in *ESEM'11: Proc. of the International Symposium on Empirical Software Engineering and Measurement*, 2011.

[38] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *ICSE'12: Proc. of the International Conference on Software Engineering*, 2012.

[39] 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 *ASE'15: Proc. of the International Conference on Automated Software Engineering*, 2015.

[40] X. Huo, M. Li, and Z.-H. Zhou, "Learning unified features from natural and programming languages for locating buggy source code," in *Proceedings of IJCAI'2016*.

[41] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *ACM SIGPLAN Notices*, vol. 49, no. 6. ACM, 2014, pp. 419–428.

[42] V. Raychev, P. Bielik, and M. Vechev, "Probabilistic model for code with decision trees," in *OOPSLA'16: Proc. of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2016.

[43] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *ASE'16: Proc. of the International Conference on Automated Software Engineering*, 2016.

[44] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," *arXiv preprint arXiv:1409.5718*, 2014.

[45] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," in *FSE'16: Proc. of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016.

[46] C. Liu, X. Chen, E. C. Shin, M. Chen, and D. Song, "Latent attention for if-then program synthesis," in *NIPS'16: Proc. of the Advances in Neural Information Processing Systems*, 2016.

[47] S. Reed and N. De Freitas, "Neural programmer-interpreters," *arXiv preprint arXiv:1511.06279*, 2015.

[48] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, "Deepcoder: Learning to write programs," *arXiv preprint arXiv:1611.01989*, 2016.

[49] C. Shu and H. Zhang, "Neural programming by example," in *AAAI'17: Proc. of the AAAI Conference on Artificial Intelligence*, 2017.

[50] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *QRS'15: Proc. of the International Conference on Software Quality, Reliability and Security*, 2015.