applied sciences

MDPI

*Article*

# An Improved CNN Model for Within-Project Software Defect Prediction

**Cong Pan [1,2,*], Minyan Lu [1,2,*], Biao Xu [1,2] and Houleng Gao [1,2]**

[1] The Key Laboratory on Reliability and Environmental Engineering Technology, Beihang University, Beijing 100191, China; xxbbzxc@163.com (B.X.); gaohouleng@126.com (H.G.)

[2] School of Reliability and System Engineering, Beihang University, Beijing 100191, China

* Correspondence: cong_pan@buaa.edu.cn (C.P.); lmy@buaa.edu.cn (M.L.)

Received: 18 March 2019; Accepted: 22 May 2019; Published: 24 May 2019

**Abstract:** To improve software reliability, software defect prediction is used to find software bugs and prioritize testing efforts. Recently, some researchers introduced deep learning models, such as the deep belief network (DBN) and the state-of-the-art convolutional neural network (CNN), and used automatically generated features extracted from abstract syntax trees (ASTs) and deep learning models to improve defect prediction performance. However, the research on the CNN model failed to reveal clear conclusions due to its limited dataset size, insufficiently repeated experiments, and outdated baseline selection. To solve these problems, we built the PROMISE Source Code (PSC) dataset to enlarge the original dataset in the CNN research, which we named the Simplified PROMISE Source Code (SPSC) dataset. Then, we proposed an improved CNN model for within-project defect prediction (WPDP) and compared our results to existing CNN results and an empirical study. Our experiment was based on a 30-repetition holdout validation and a 10 * 10 cross-validation. Experimental results showed that our improved CNN model was comparable to the existing CNN model, and it outperformed the state-of-the-art machine learning models significantly for WPDP. Furthermore, we defined hyperparameter instability and examined the threat and opportunity it presents for deep learning models on defect prediction.

**Keywords:** CNN model; within-project defect prediction; abstract syntax tree; deep learning; hyperparameter instability

## 1. Introduction

The increasing complexity of modern software has elevated the importance of software reliability. Building highly reliable software requires a considerable amount of testing and debugging. However, since both budget and time are limited, these efforts must be prioritized for efficiency. As a result, software defect prediction techniques, which predict the occurrence of bugs, have been widely used to assist developers in prioritizing their testing and debugging efforts [1].

Software defect prediction [2–5] is the process of building classifiers to predict whether defects exist in a specific area of source code. The prediction results can assist developers in prioritizing their testing and debugging efforts. From the perspective of prediction granularity, software defect prediction can include method-level, class-level, file-level, package-level, and change-level defect prediction. For the current research we focused on file-level defect prediction, because there exists ample amount of labeled data. Typical software defect prediction consists of two phases [6]: (1) extracting features from software artifacts such as source code, and (2) building classification models using various machine learning algorithms for training and validation.

Previous research on software defect prediction has taken one of two research directions: one is creating new features or using different combinations of existing features to better characterize defects and the other is utilizing existing statistics-based models or machine learning models and

making improvements to existing models for better classification of buggy code. Alongside the first direction, researchers have designed various hand-crafted features to capture defect characteristics, for example, Halstead features [7] based on operator and operand counts, McCabe features [8] based on dependencies, and other comprehensive feature sets including Chidamber and Kemerer (CK) features [9], metrics for object-oriented design (MOOD) features [10], and code change features [11]. As for the second direction, many machine learning models have been designed to target software defect prediction, including the decision tree [12], random forest [13], logistic regression [14], naive Bayes [15], and dictionary learning [16] models.

Unlike natural languages, programs have a well-defined syntax in the form of the Backus-Naur Form, which can be further organized into abstract syntax trees (ASTs) [17]. ASTs have been widely used for characterizing source code [18,19]. However, program semantics is also buried deep in ASTs [20], however, it is ignored by traditional hand-crafted features. Researchers have shown the usefulness of ASTs in software engineering tasks such as code completion and bug detection [17–19,21,22]. In recent studies [23,24], researchers have begun to extract features from ASTs to capture both syntax and semantics of source code, and these features have shown great improvements over traditional hand-crafted features.

Since the advent of AlexNet [25] in 2012, deep learning has been widely used in areas such as image recognition, speech recognition, and natural language processing [26]. It has emerged as a powerful technique for automated feature generation since deep learning architecture can effectively capture highly complicated nonlinear features [26]. To make use of its powerful feature generation ability, the deep belief network (DBN) [23] was proposed. One year later, the state-of-the-art method [24] leveraging convolutional neural network (CNN) was proposed to learn the semantics of source code via AST programs, which have been shown to outperform traditional feature-based approaches in software defect prediction.

However, whether the CNN model outperforms traditional machine learning models has remained unclear. First, the dataset used in the CNN study [24] was rather small. There were only 14 versions of seven projects, which likely added to the work's statistical uncertainty. Second, the results in the CNN paper were derived from insufficiently repeated experiments. As suggested by Arcuri [27], to minimize statistical bias and variance, there should be no fewer than 30 repeated experiments, and research on model validation by Tantithamthavorn [28] also supported that conclusion. The CNN study repeated the experiment only 10 times, a choice that undermined the confidence in their conclusions. Lastly, the researchers for the CNN study did not choose the proper traditional baseline. The traditional baseline [29] in the CNN study included only a logistic regression model for within-project defect prediction (WPDP) and this was not suitable, as no research indicates that logistic regression could achieve best performance for WPDP.

For this study, we proposed an improved CNN model for software defect prediction to validate the hypothesis that CNN models can outperform traditional machine learning models for WPDP. Specifically, we first compiled source code to ASTs and followed existing strategies to select them and form token vectors such that the token vectors act as a summarization of source files. Then, we converted the code vector strings into integers, which were then sent to our CNN model as input. Our CNN model comprised a word embedding layer, three convolutional layers, three max-pooling layers, four dense layers, and three dropout layers. Finally, the generated features produced by the CNN model were fed into a logistic regression classifier to predict whether a source file was buggy. We first evaluated our method on the Simplified PROMISE Source Code (SPSC) dataset, which was used in a paper by Li [24], to confirm that our CNN model was comparable to or better than existing state-of-the-art models in terms of F-measure (also known as F1 score). Then, we evaluated our model on the PROMISE Source Code (PSC) dataset, which was derived from the PROMISE repository [30], to target AST-based features in terms of F-measure, G-measure, and Matthews correlation coefficient (MCC). The experimental results indicated that our model was comparable to the existing CNN model, and outperformed various traditional machine learning models including the random forest, naïve Bayes, logistic gegression, RBF network and decision tree models for WPDP. Furthermore, we discovered that different hyperparameter settings of a CNN model could generate similar average

results for all projects but is markedly different in terms of individual project or version, which we referred to as hyperparameter instability. We concluded that hyperparameter instability may present a threat and an opportunity for deep learning models on defect prediction.

Our study made the following contributions:

- We proposed an improved CNN model for better generalization and capability of capturing global bug patterns. The model learned semantic features extracted from a program's AST for defect prediction.
- We built and published a dataset named PSC, which targeted AST-based features from the PROMISE repository based on five principles. The PSC dataset was larger than the SPSC dataset, and we excluded versions for which source code could not be found, or the labeled CSV file did not match the source code.
- We performed two experiments to validate that the CNN model could outperform the state-of-the-art machine learning models for WPDP. The first experiment demonstrated the validity of our improved model, while the second experiment validated the performance of our improved CNN model as compared with other machine learning models.
- We performed a hyperparameter search, which took dense layer numbers, dense layer nodes, kernel size, and stride step into consideration to uncover empirical findings.
- We concluded that hyperparameter instability may be a threat and an opportunity for deep learning models on defect prediction. The discovery was based on experimental data and targeted deep learning models like CNN.

## 2. Background

### *2.1. Software Defect Prediction*

Figure 1 presents a typical file-level software defect prediction process, which is the prediction process adopted in most recent studies. As the figure shows, the first step of the process is to extract program modules (i.e., source files) from software repositories. The second step is to label program modules as buggy or clean. The labeling criteria are based on post-release defects collected from bug fixing information or bug reports in bug tracking systems (i.e., Bugzilla). A source file is labeled buggy if it contains at least one bug. Otherwise, it is labeled clean. The third step is to extract features from the labeled program modules to form training instances. Traditional features consist of code metrics such as Halstead features [7], McCabe features [8], and CK features [9], and process metrics such as change histories. Recently, AST-based features have also been presented [23,24]. The fourth step is to build a classification model and use training instances to train the model. Researchers have proposed various machine learning models, such as the decision tree [12], random forest [13], logistic regression [14], naive Bayes [15], and dictionary learning [16] models, as well as several deep learning models including DBN [23] and CNN [24]. The last step is to feed new program feature instances into the trained classifier to predict whether a source file is buggy or clean.

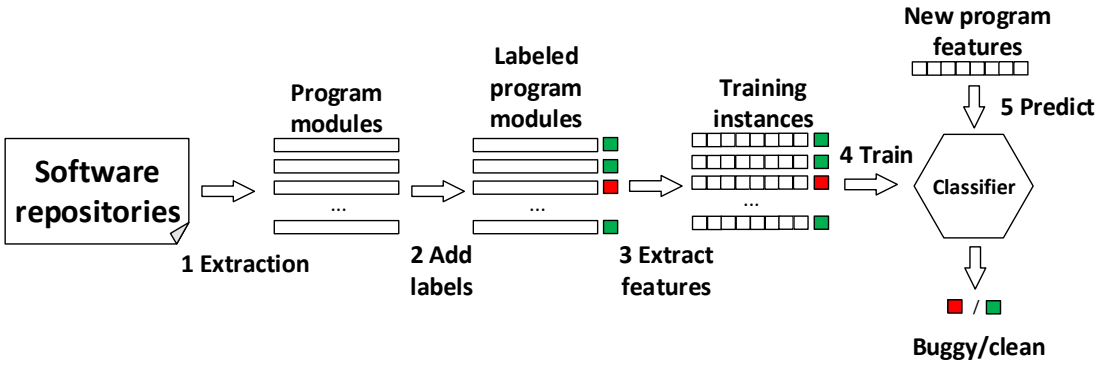


**Figure 1.** A typical software defect prediction process based on machine learning.

In the following we will explain some terminologies regarding software defect prediction. A **training set** refers to a set of instances used to train a model, whereas a **test set** refers to a set of instances used to evaluate the learned model. When performing **WPDP**, the training set and the test set come from the same project. WPDP can be performed if defect information regarding current version or previous versions can be obtained. More specifically, in **within-version WPDP** the training set and the test set come from the same version of the same project, whereas, when performing **cross-version WPDP,** the training set and the test set come from different versions of the same project. We focused on WPDP in this study to compare our improved model with existing baseline methods [23,24,29,31]. Specifically, we performed cross-version WPDP on the SPSC dataset in the first experiment, and within-version WPDP on the PSC dataset in the second experiment.

### *2.2. Convolutional Neural Network*

A CNN is a special type of neural network used to process data that has a known, grid-like topology [26]. Examples include one-dimensional (1D) time-series data as well as two-dimensional (2D) and three-dimensional (3D) image data. CNN has been tremendously successful in practical applications, including speech recognition [32], image classification [25], and natural language processing [33,34]. In this work, we leveraged a CNN model to extract features from AST programs automatically.

Figure 2 demonstrates the overall architecture of a CNN. A CNN consists of convolutional layers and pooling layers. In a simple fully-connected network, which we call dense network, neuron units are connected to all neuron units of its neighboring layers. In CNN, neural units connected to these two layers are sparsely connected, which is determined by kernel size and pooling size. A CNN network features two key characteristics that are sparse connectivity and shared weights. These two characteristics help reduce model capacity and capture global patterns rather than local patterns.

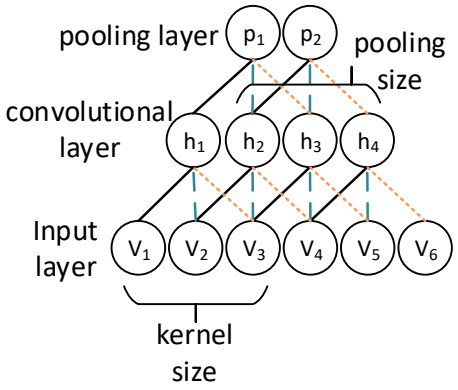


**Figure 2.** A basic convolutional neural network (CNN) architecture.

Sparse connectivity means that each neuron is connected to only a limited number of other neurons [35]. In a CNN, sparse connectivity is controlled by kernel size and pooling size. Let us take node $V_3$ in Figure 2 as an example. When kernel size = 3, it is connected to only three nodes $h_1$, $h_2$, and $h_3$, whereas, $h_4$ is not affected by $V_3$. In the same way, $h_2$ is affected only by $V_2$, $V_3$, and $V_4$. Each subset connecting to the next layer in the CNN is called a local filter, which captures a specific kind of patterns. To calculate the output to the next layer, one can multiply each local filter by outputs from the previous layer, add a bias, and then perform a nonlinear transformation. In Figure 2, if we denote the $i^{th}$ neuron in the $m^{th}$ layer (convolutional layer) as $h_i^m$, the weights of the $i^{th}$ neuron in the $(m\text{-}1)^{th}$ layer as $W_i^{m-1}$, the bias in the $(m\text{-}1)^{th}$ layer as $b^{m-1}$, and we use ReLU as our activation function, then the output can be calculated as follows:

$$h_i^m = ReLU(W_i^{m-1} * V_i^{m-1} + b^{m-1}). \quad (1)$$

Shared weight means that each filter shares the same parameterization (weight vector and bias) [24]. For example, in Figure 2 all the solid black lines linking the input layer and convolutional layer

share the same parameters, and the same is true of the blue sparse-dotted lines and the orange dense-dotted lines. Shared weight enables a CNN to capture features independent of their positions and it can also effectively reduce model capacity.

In CNN, a pooling layer is usually used after a convolutional layer. It replaces the output of the network at a certain location with a summary statistic of the nearby outputs [35]. Pooling helps to make representations more robust, in other words, approximately invariant to small changes in the input. Moreover, it can reduce the numbers of intermediate representations which translates into reduced model capacity.

There are many hyperparameters in CNN models including filter size, pooling size, and convolutional and pooling layer numbers. These hyperparameters must be tuned to make the model work well. We will discuss the results of our hyperparameter tuning process in section 5.3.

### *2.3. Word Embedding*

The key idea behind word embedding is distributed representation of words. When a one-hot encoder is used, each word is encoded into a vector in the shape of {0…0, 1, 0…0}. The length of the vector is equal to the vocabulary size N, and words are independent of each other. On the contrary, distributed representation regards words as related items, and a word is encoded into a shorter vector. The length of the vector is far less than the vocabulary size N, and each item is a decimal number between 0 and 1. After such word embeddings, the distance of words can be calculated easily.

In our improved CNN model, we built an embedding layer as the first layer, which was based on the skip-gram model. Detailed information on the mathematical formulations of the skip-gram model is given in [36]. In Figure 3 we illustrate the very basics of a skip-gram model. First, each word in the corpus is selected, and its surrounding words (define by window size) are picked to form tuples with two elements. A skip-gram model focuses on one word ("new" for example) as input, which is represented as a one-hot encoder. The word vector for the input word is learned in the hidden layer, and the length of the word vector is far less than the word vocabulary size. In the output layer, the model predicts the neighboring words of the input word using softmax classifier. The weights of the hidden layer are updated by comparing the model prediction result (neighboring word within word vocabulary) with the neighboring word extracted from the original corpus ("york" for example). After training, the learned word vectors can be used to represent input words, and they are extracted as output of the embedding layer.

In software defect prediction, input vectors representing source code ASTs information are embedded, which can be measured by distance and features a grid-like property. CNN models work perfectly on grid-like data, and that is why the CNN model can be utilized for defect prediction after word embedding.

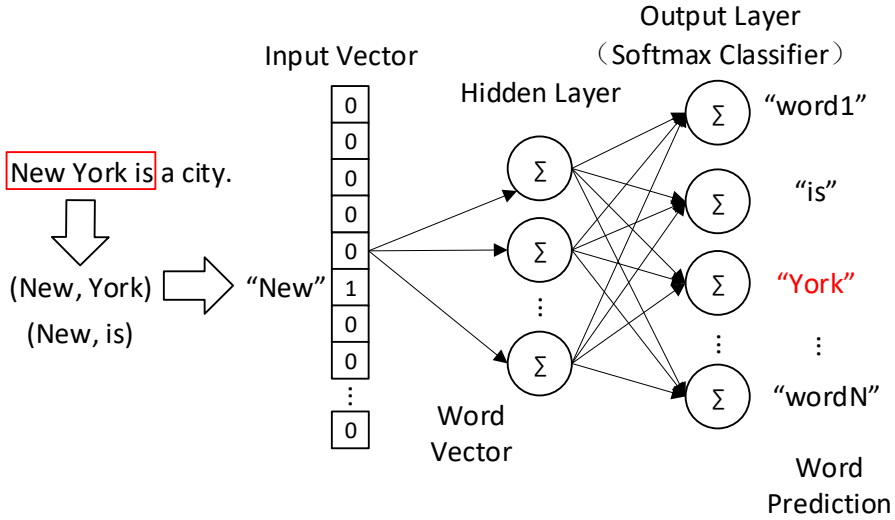


**Figure 3.** A skip-gram model.

### *2.4. Dropout*

Dropout [37] is a network model aiming to deal with model overfitting issues. Its key idea is to randomly drop neural units as well as their connections during training, which would prevent complex co-adaptions of units and lower model generalization error (differences between model performance on training set and test set). The concept of co-adaptation was proposed by Srivastava et al. [37]. In the backward propagation process of a neural network, the weights of a unit are updated given what other units are doing, so the weight might be updated to compensate for the mistake of other units, which is called co-adaptation. When adding dropout layers, a unit is unreliable because it may be randomly dropped. Therefore, each unit would learn better features instead of fixing the mistakes of other units. Weight rescaling is performed on test sets to compensate for the dropped units.

Refer to Figure 4 as an example to illustrate the mechanics of dropouts. When we add a dropout layer for the hidden layer and set the dropout probability to 0.5, hidden units are randomly chosen to be dropped at a probability of 0.5, as $H_2$ and $H_4$, for example. The connections between the two nodes and the input/output layers are also dropped. In this case, the weight update of $H_1$ and $H_3$ would be independent of $H_2$ and $H_4$, which would prevent co-adaptation of units and lower model generalization error.

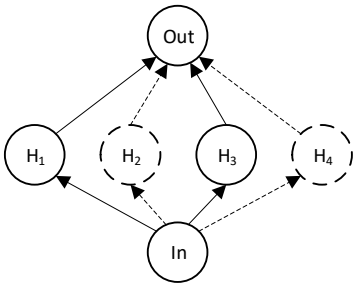


**Figure 4.** An example of dropout when p = 0.5.

In our improved CNN model, we used dropout between dense layers in order to better model generalization. Although dropout could also be used between convolutional layers and max-pooling layers, we did not add dropouts because in convolutional layers and max-pooling layers units are already sparsely connected, and because adding more sparsity through dropout would not be necessary.

## 3. Approach

In this section, we propose our improved CNN model, which followed the framework proposed by Li [24]. Figure 5 shows the overall workflow of a software defect prediction based on our improved CNN model.

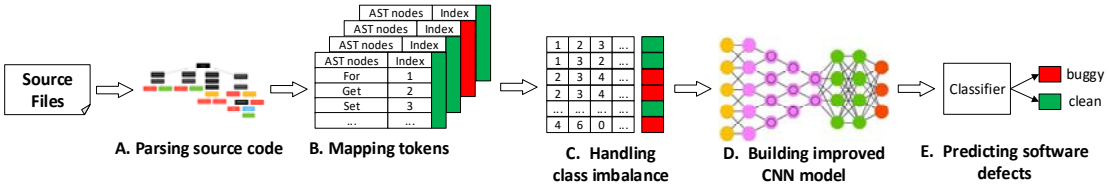


**Figure 5.** The overall workflow of a software defect prediction based on our improved CNN model.

As Figure 5 shows, the prediction process consists of five parts. First, we parsed source codes into ASTs and generated token vectors by selecting specified AST node types described in Figure 6. Next, we mapped the string token vectors into integer input vectors to the CNN model. Then, we utilized random over-sampling methods to handle class imbalance problems. Finally, we built our improved CNN model to predict software defects.

### *3.1. Parsing Source Code*

Since we used source code as input, a suitable code representation form was deemed to be beneficial for parsing source code. Code representations include character-level, token-level, AST-node-level, tree-level, graph-level, path-level, among others. The AST-node-level code representations have proven to beat character-level, token-level, and representations of higher granularities in program classification tasks [38], so we extracted AST nodes from Java source code.

According to Li [24], three types of AST nodes are extracted as tokens: (1) nodes of method invocations and class instance creations, which are represented as method names or class names; (2) declaration nodes such as method declarations, type declarations, and enum declarations, which are represented by their values; and (3) control-flow nodes including while statements, catch clauses, if statements, and throw statements, which are represented by their node types. We excluded certain node types including assignment and package declaration, because (1) information in the node could not be traced, as in package declaration; (2) there was such little information in the node that a huge amount of labeled data would have been needed to train such a model, as in assignment; (3) the frequency of the node was too low, as in EnhancedForControl. A full list of the selected 29 AST node types is listed in Figure 6.

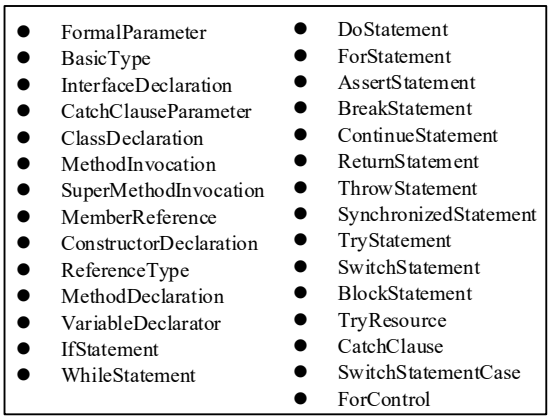


**Figure 6.** Selected abstract syntax tree (AST) node types.

To parse source files into ASTs, we utilized a python tool called javalang. It enables parsing of Java 8, and it is based on the official Java language specification. Due to the limited functionality of javalang, several Java source files could not be parsed correctly, which may have hampered data preprocessing. We considered three strategies to solve the problem: (1) correct the source file grammar so that javalang could parse, (2) delete part of the source code that could not parse, (3) delete the file directly. We adopted the third strategy for simplicity.

A motivating example of Java code for add calculation is shown in Figure 7. After parsing code by javalang and traversing the AST, a list including 39 elements was generated, which starts with CompilationUnit and ends with MemberReference. After selection of the 39 nodes, a refined list including 26 elements was generated, which starts with ClassDeclaration and ends with MemberReference. Some node types, such as CompilationUnit, Assignment and BinaryOperation were excluded according to the criteria listed above. Next, we replaced some of the node types with its names. For example, we replaced the fourth node ReferenceType with String to express detailed information. In the end, a list including 26 elements was generated, the first five elements of which were CommandParameter, main, args, String, and int.

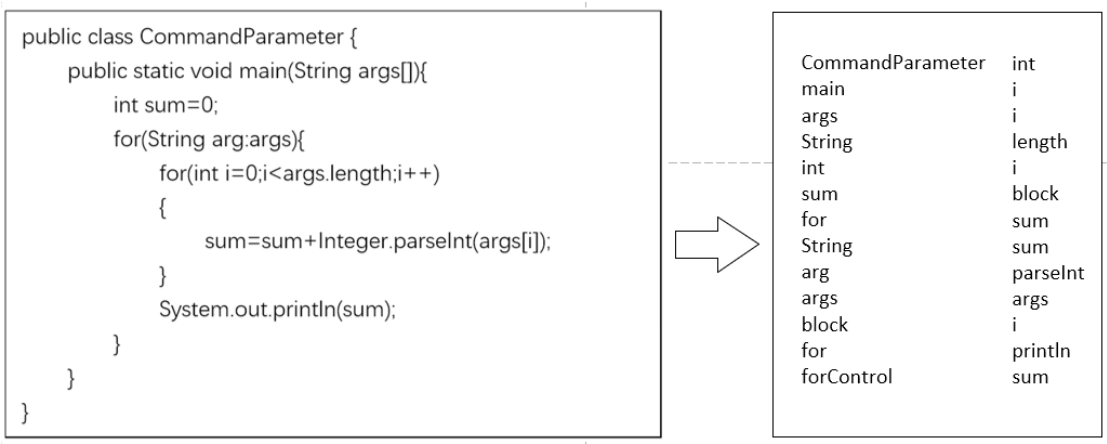


**Figure 7.** Example of Java code for add calculation and its parsed abstract syntax tree (AST) tokens.

### *3.2. Mapping Tokens*

After parsing source code, we obtained a token vector for each source file. However, these token vectors could not serve as the direct input for a CNN model, and therefore we first needed to map tokens from strings to integers. Then, we carried out a conversion that mapped each string token to an integer ranging from one to the total number of token types so that each different string was represented by a unique integer all the time. In addition, the CNN model requires input vectors to have equal length. However, the length of the input vectors varied by the number of extracted AST nodes for each source file after conversion. To solve the problem, we appended zero to the integer vectors to make their lengths equal to the longest vector. The digit zero would not affect mapping spaces because the mapping started from one.

Following common practices in the natural language processing (NLP) domain, we deleted infrequent tokens that might have been used for a specific reason and may not have been suitable for training instances. We define infrequent as once or twice, and these infrequent tokens were mapped to zero.

### *3.3. Handling Class Imbalance*

Perfectly balanced datasets rarely exist in real life, so we had to face class imbalance problems when we trained the models for software defect prediction. Imbalance means that the number of buggy samples was not proportional to the number of clean samples. The class imbalance problem generally decreases the performance of classification models.

We considered three strategies to solve the problem: (1) under-sampling, which means reducing the number of samples from the majority class; (2) over-sampling, which means increasing the number of samples from the minority class; and (3) a combination of over- and under-sampling, which aims to get more robust results. Because our inputs were categorical data, some powerful methods like SMOTE and its variations could not be used. Thus, we used random over-sampling for simplicity. During random over-sampling, existing buggy and clean files were duplicated until their ratio reached 50:50. Handling the class imbalance applied to the training set only.

### *3.4. Building the Improved CNN Model*

We built our improved CNN model based on common practices in other fields [25,33]. The overall architecture is shown in Figure 8. Our improved CNN model consisted of an embedding layer, three convolutional layers and max-pooling layers to extract global patterns, a flattening layer, four dense layers with dropouts to generate deep features and help better generalization, and finally, a logistic regression classifier to predict whether a source file was buggy.

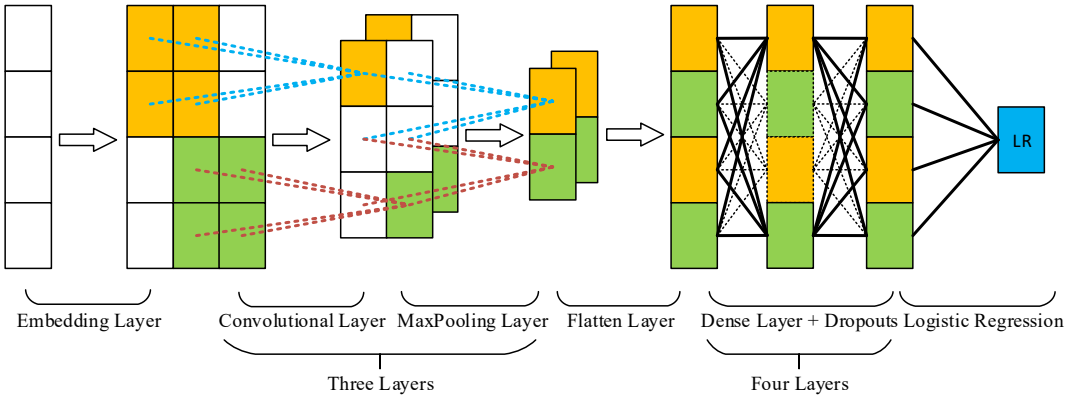


**Figure 8.** Illustration of the proposed improved CNN model.

Other detailed architecture information is listed below.

- Implementation framework: We use Keras (https://www.tensorflow.org/guide/keras), a high-level API based on tensorflow to build our model for simplicity and correctness. The version of tensorflow/keras is 1.8.
- Word embeddings: Note that the implementation of the word embedding layer is also based on Keras; it was wrapped inside the CNN model. Although word embedding is not a part of CNN mathematically, we regarded it as our first layer in our CNN model.
- Three convolutional layers and max-pooling layers: It is universally accepted that increasing the depth of deep models could get better results. Thus, we increased numbers of convolutional layers and max-pooling layers from 1 to 3. Because adding more such layers requires the output of the embedding layer to increase accordingly, which is time-consuming, we did not make further attempts.
- Activation function: Except for the last layer which used the sigmoid activation function for classification, all other layers used the rectified linear unit (ReLU) activation function.
- Parameter initialization: Due to limited calculation resources, large epochs like 100 were not set for training a model. Therefore, it was essential to speed-up model training during the initial epochs. We used He_normal [39] to initialize embedding layers, convolutional layers, and dense layers due to its high efficiency in loss decrease. For the last layer, we used Glorot_normal [40] which targets the sigmoid activation function for initialization.
- Dropouts: We added dropouts between dense layers, which is a common practice to prevent model overfitting [37].
- Regularization: To avoid model overfitting and make the model converge more quickly, we added L2 regularizations for embedding layer and dense layers. When L2 is used, an extra L2 norm of weights is added to the loss function to decrease model capacity, which is essential for model generalization.
- Training and optimizer: Our model trained by using the mini-batch stochastic gradient descent (SGD) algorithm [41], along with an Adam optimizer [42], in order to get across the saddle points in the hyperplanes to speed up training. We used binary cross-entropy as the loss function.
- Hyperparameters: We used the best hyperparameter combinations for both experiments on SPSC and PSC dataset, which is illustrate in section 5.3.

Architectural comparison between our improved CNN model and the CNN model [24] is summarized in Table 1. Parameters listed above apply to the experiments on the PSC and SPSC datasets.

**Table 1.** Architectural comparison between Li's CNN model and our improved CNN model. Differences are highlighted in bold.

| | **Li's CNN** | **Improved CNN** |
|---|---|---|
| Embedding layer | Yes | Yes |
| #Convolutional Layers | 1 | **3** |
| #Pooling layers | 1 | **3** |
| Activation function | ReLU + sigmoid (last dense layer) | ReLU + sigmoid (last dense layer) |
| Parameter initialization | None | **He_normal (embedding layer, convolutional layer, dense layers) Glorot_normal (last dense layer)** |
| Dropouts | None | **Between dense layers** |
| Regularization | None | **L2 Regularization (embedding layer, dense layers)** |
| Training and optimizer | Mini-batch SGD + Adam (loss function not given) | Mini-batch SGD + Adam + binary cross-entropy |

### *3.5. Predicting Software Defects*

We employed a logistic regression model as our classifier in compliance with previous work [24] because it was easy to implement in Keras and we focused on the overall deep learning models rather than the last layer of the CNN model as classifiers. After preprocessing of the labeled source files, we split the dataset into a training set and a test set following a split ratio of 80:20. We performed a stratified split to ensure that the ratio remained unchanged in both the training test and the test set. After we fed the training set to our improved CNN model, all parameters, including weights and biases, were fixed. Then, for each file in the test set, we ran the trained CNN model to obtain our prediction results. The results were in the form of a decimal number between zero and one, based on which we predicted a source file as buggy or clean. If the result was above 0.5, the prediction was regarded as buggy; otherwise it was regarded as clean.

## 4. Experimental Setup

All of our experiments were run on a Linux server with Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz. We used CPUs for training deep learning models.

### *4.1. Evaluation Metrics*

To measure the defect prediction results, we used three metrics: F-measure, G-measure, and MCC under different experiment settings. These metrics have been widely used in various software defect prediction studies [5,23,24,31,43,44].

F-measure is the harmonic mean of precision and recall. It ranges from 0 to 1. A higher F-measure means a better prediction performance. We calculated the F-measure as follows:

$$precision = \frac{TP}{TP+FP}. \quad (2)$$

$$recall = TPR = \frac{TP}{TP+FN}. \quad (3)$$

$$\text{F-}measure = \frac{2*precision*recall}{precision+recall} \quad (4)$$

G-measure is the harmonic mean of true positive rate (TPR, also recall) and true negative rate (TNR). It aims to measure the bad influences of false alarms on software defect prediction. G-measure ranges from 0 to 1. A higher G-measure means a better prediction performance. We calculated the G-measure as follows:

$$TNR = \frac{TN}{TN+FP} \quad (5)$$

$$\text{G-}measure = \frac{2*TPR*TNR}{TPR+TNR}. \quad (6)$$

MCC describes the correlation between true values and predicted values. It ranges from –1 to 1. An MCC of 1 means perfect prediction, 0 means random prediction, and –1 means that all predictions failed. We calculated the MCC as follows:

$$MCC = \frac{TP*TN-FP*FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}. \quad (7)$$

### *4.2. Evaluated Projects and Datasets*

We utilized the dataset from the PROMISE Repository (The original website is http://openscience.us/repo; however, the website is currently unavailable, and a backup repository can be found at https://github.com/opensciences/opensciences.github.io), which is a publicly available repository for domains including software defect prediction. For this study, we selected two subsets of the original dataset which were the SPSC dataset and the PSC dataset. The SPSC dataset was a smaller dataset that was used in an existing CNN paper [24], while the PSC dataset was a larger dataset designed by hand for defect prediction of a wider range. Both datasets included the version numbers, the class name of each file, and the defect label for each source file, as well as 20 traditional features such as weighted methods per class (WMC), depth of inheritance tree (DIT), and number of children (NOC). Since we used the AST node information as input, we downloaded the corresponding versions of the projects from open source repositories rather than using traditional features.

In the following we describe the detailed design of the PSC dataset. To facilitate feature extraction from the source code ASTs, we set five guidelines for dataset design: (1) the dataset had to come from existing high-quality defect prediction repositories, (2) the dataset had to be based on open source projects, (3) the projects included in the dataset had to contain at least two versions for cross-version comparison, (4) the link between the open-source project versions and labeled CSV files had to be verified to make comparisons with deep features and traditional features easier, and (5) the dataset had to exclude extremely abnormal buggy rate versions of projects to correctly calculate evaluation metrics. There are various open repositories for defect prediction, such as PROMISE [30], AEEEM [43], NASA MDP [45], NETGENE [46], and RELINK [47]. Some of the repositories are based on closed-source software, such as NASA MDP. Although there exists labeled data as well as hand-crafted features in these repositories, AST-based features cannot be extracted without open source code. Therefore, we chose to select projects and versions from the PROMISE repository. Next, we chose projects that included multiple versions from the PROMISE repository. When checking links between open source project versions and labeled CSV files, we selected all similar versions of the source code (i.e., 1.4 and 1.4.4) to prevent mistakes in the mapping process. After the match and validation process, we found that in Xerces 1.4 the labeled CSV file did not match the source code versions in nearly half of the instances, so we excluded Xerces 1.4 from our PSC dataset. In the last step, we did not exclude project versions with less than 100 instances or a buggy rate above 95% or below 5% because these projects could still prove helpful if the size of the dataset was not too small. Instead, we chose to exclude Xalan 2.7 because its buggy rate was at 98.8%, which would cause problems in evaluation metric calculations, as the test set may hold no clean samples.

Tables 2 and 3 provide detailed information on the SPSC and PSC datasets. Table 2 includes 6538 files from 14 versions of seven projects, and information including the previous version (Vp) and the newer version (Vn), average files, and the average buggy rate. We used Vp to train the model, Vn to validate the model, and performed 30-holdout repeated experiments, which follows the cross-version WPDP pattern.

**Table 2.** Simplified PROMISE Source Code **(**SPSC) dataset description. SPSC is applied for cross-version within-project defect prediction (WPDP). Numbers in brackets indicate numeral changes compared with the original dataset from PROMISE repository.

| Project | Versions (Vp, Vn) | #Files | #Defects | Avg. Buggy Rate (%) |
|---|---|---|---|---|

| | | | | |
|---|---|---|---|---|
| Camel | 1.4, 1.6 | 1781(–6) | 333(0) | 18.7(+0.6) |
| jEdit | 4.0, 4.1 | 547(–71) | 134(–20) | 24.5(–0.4) |
| Lucene | 2.0, 2.2 | 420(–22) | 234(–1) | 55.7(+2.5) |
| Xalan | 2.5, 2.6 | 1629(–59) | 790(–8) | 48.5(+1.2) |
| Xerces | 1.2, 1.3 | 882(–11) | 138(–2) | 15.6(–0.1) |
| Synapse | 1.1, 1.2 | 461(–17) | 141(–5) | 30.6(+0.1) |
| Poi | 2.5, 3.0 | 818(–9) | 529(–9) | 64.7(+0.7) |
| Total | - | 6538(–245) | 2299(–45) | 35.2(+0.6) |

Table 3 includes 14,066 files from 41 versions of 12 projects, and information including version, numbers of files, and buggy rate. From the perspective of file numbers, it can be observed that the SPSC dataset has expanded more than twice in size as compared with the SPSC dataset. We used the 10 × 10 cross-validation strategy for training and validation, which followed the within-version WPDP pattern.

The SPSC and PSC dataset are available at https://github.com/penguincwarrior/CNN-WPDP-APPSCI2019.

**Table 3.** PROMISE Source Code (PSC) dataset description. PSC is applied for within-version WPDP. Numbers in brackets indicate numeral changes compared with the original dataset from PROMISE repository.

| Project | Version | #Files | #Defects | Buggy Rate (%) |
|---|---|---|---|---|
| Ant | 1.3 | 124(–) | 20(0) | 16.0(+0.1) |
| | 1.4 | 177(–1) | 40(0) | 22.6(+0.1) |
| | 1.5 | 278(–15) | 29(–3) | 10.4(–0.5) |
| | 1.6 | 350(–1) | 92(0) | 26.3(+0.1) |
| | 1.7 | 741(–4) | 166(–1) | 22.4(0) |
| Camel | 1.0 | 339(–0) | 13(0) | 3.8(0) |
| | 1.2 | 595(–13) | 216(0) | 36.3(+0.8) |
| | 1.4 | 847(–25) | 145(0) | 17.1(+0.5) |
| | 1.6 | 934(–31) | 188(0) | 20.1(+0.6) |
| Ivy | 1.1 | 111(0) | 63(0) | 56.8(0) |
| | 1.4 | 241(0) | 16(0) | 6.6(0) |
| | 2.0 | 352(0) | 40(0) | 11.4(0) |
| JEdit | 3.2 | 260(–12) | 90(0) | 34.6(+1.5) |
| | 4.0 | 281(–25) | 67(–8) | 23.8(–0.7) |
| | 4.1 | 266(–46) | 67(–12) | 25.2(–0.1) |
| | 4.2 | 355(–12) | 48(0) | 13.5(+0.4) |
| | 4.3 | 487(–5) | 11(0) | 2.3(0) |
| Log4j | 1.0 | 119(–16) | 34(0) | 28.8(+3.4) |
| | 1.1 | 104(–5) | 37(0) | 35.6(+1.6) |
| | 1.2 | 194(–11) | 186(–3) | 95.9(+3.7) |
| Lucene | 2.0 | 186(–9) | 91(0) | 48.9(+2.3) |
| | 2.2 | 234(–13) | 143(–1) | 61.1(+2.8) |
| | 2.4 | 330(–10) | 203(0) | 61.5(+1.8) |
| Pbeans | 1.0 | 26(0) | 20(0) | 76.9(0) |
| | 2.0 | 51(0) | 10(0) | 19.6(0) |
| Poi | 1.5 | 235(–2) | 141(0) | 60.0(+0.5) |
| | 2.0 | 309(–5) | 37(0) | 12.0(+0.2) |
| | 2.5 | 380(–5) | 248(0) | 65.3(+0.8) |
| | 3.0 | 438(–4) | 529(0) | 64.2(+0.6) |
| Synapse | 1.0 | 157(0) | 16(0) | 10.2(0) |

| | | | | |
|---|---|---|---|---|
| | 1.1 | 205(–17) | 55(–5) | 26.8(–0.2) |
| | 1.2 | 256(0) | 86(0) | 33.6(0) |
| Velocity | 1.4 | 195(–1) | 147(0) | 75.4(+0.4) |
| | 1.5 | 214(0) | 142(0) | 66.4(0) |
| | 1.6 | 229(0) | 78(0) | 34.1(0) |
| Xalan | 2.4 | 676(–47) | 110(0) | 16.3(+1.1) |
| | 2.5 | 754(–49) | 379(–8) | 50.3(+2.1) |
| | 2.6 | 875(–10) | 411(0) | 47.0(+0.5) |
| Xerces | Initial | 162(0) | 77(0) | 47.5(0) |
| | 1.2 | 436(–4) | 70(–1) | 16.1(–0.1) |
| | 1.3 | 446(–7) | 68(–1) | 15.2(0) |
| Total | - | 14066(–289) | 6542(–77) | 31.4(+0.6) |

### *4.3. Baseline Methods*

We compared our improved CNN method on the SPSC dataset with the following baseline methods:

- CNN [24]: This is a state-of-the-art method of deep software defect prediction that extracts AST node information as model input, and it proposes deep models (CNN) combined with word embeddings for defect prediction.
- DBN [23]: This was the first method of deep software defect prediction conceived. It extracts AST node information as model input, and it proposes deep models (DBN) for defect predictions. We used the results from the CNN paper [24] for easy comparison.
- Traditional (LR) [29]: This is an empirical result of traditional software defect prediction, which utilizes 20 code metrics extracted manually in the PROMISE repository as model input, and it proposes logistic regression models for defect prediction. We also used the results from the CNN paper for ease of comparison. It should be noted that the traditional model [29] is not the most suitable baseline to represent traditional machine learning models.

We then compared our improved CNN method on the PSC dataset with the following baseline methods:

- Five machine learning models [31]: Five models were trained and validated in the within-version WPDP pattern. These five models were the decision tree (DT), logistic regression (LR), naïve Bayes (NB), random torest (RF), and RBF network (NET). These models are the most prevalent machine learning models, which in general represent the top results for WPDP. We excluded the SVM model because it tends to work poorly in empirical scenarios [31].
- RANDOM [31]: This model predicts a source file as buggy at the probability of 0.5. If the RANDOM model beats any model on averaged metrics, it means that the model is worse than random guessing.
- FIX [31]: This model predicts all source files as buggy, which works as a null hypothesis model. If the FIX model beats any model, it means that the model should not be used in practical situations.

In the original empirical paper [31], no detailed experimental data regarding each version were found. Therefore, we obtained access to the corresponding Github pages where detailed raw experiment data were available. We then filtered the experiment data to fit our PSC dataset and calculated the average results from multiple experiments. The Github website is as follows: https://github.com/sherbold/replication-kit-tse-2017-benchmark.

### *4.4. Model Validation Strategy*

To reduce statistical bias and variance, we adopted the idea that the experiment should be repeated no less than 30 times [27]. For better comparison of respective baselines, we performed 30-holdout repeated experiments on the SPSC dataset and 10 × 10 cross-validation experiments on the PSC dataset. A 30-holdout repeated experiment means that the same model is trained and validated

30 times, and the experimental hyperparameters remain unchanged except for random seeds. A 10 * 10 cross-validation means that for each of the ten trials, the source files for each version were divided evenly into ten folds. For each trial, one fold is used for validation, and the other nine folds are used for training.

### *4.5. Statistical Hypothesis Test*

Comparing model results by simply comparing average performance metrics can be misleading due to their inherent variances. Therefore, statistical hypothesis tests were essential in our study.

We adopted the Friedman test [48] to check whether there were statistically significant differences between the models. The Friedman test aims to check whether the null hypothesis that all models perform equally can be rejected. It is a nonparametric statistical test using only ordinal information, so it makes no assumptions about the distributions of the prediction results.

Once the null hypothesis that there were no statistically significant differences among the models was rejected, we used the post-hoc Holm–Bonferroni test [49] afterward to check whether the performance of the two models was significantly different. The Holm–Bonferroni test checks whether two models are significantly different in terms of their ranks for all possible model pairs. It can reduce type I errors (false positives) due to the multiple hypothesis checks involved.

For both the Friedman test and the post-hoc Holm–Bonferroni test, we used a significance threshold $p = 0.05$. A calculated p-value below the threshold meant that the two methods were significantly different statistically.

In our paper, the statistical hypothesis tests were calculated on http://astatsa.com/FriedmanTest/. We have verified the results of the website via generated R code, and the answer is correct.

## 5. Results and Discussion

This section presents our experimental results. First, we propose our research questions, which were answered by subsequent experimental data and analysis.

### *5.1. Research Questions*

- **RQ1: Does our improved CNN model outperform state-of-the-art defect prediction methods in WPDP?**
- **RQ2: How does the improved CNN model perform under different hyperparameter settings?**
- **RQ3: Could our model generate results comparable to those of other baseline methods for each version of a project?**

### *5.2. Performance of the Improved CNN Model*

**RQ1: Does our improved CNN model outperform state-of-the-art defect prediction methods in WPDP?**

To better answer the research question, we broke it down into two smaller research questions.

**RQ1a: Does our improved CNN model outperform baseline methods on SPSC dataset for cross-version WPDP?**

To answer this question, we performed experiments on the SPSC dataset to pit the performance of our improved CNN model against the baseline methods, including traditional, DBN, and CNN. We followed the cross-version WPDP pattern for the experiment.

Table 4 shows the experimental results. Let us take Xalan for example. We used Xalan 2.5 for training, and a newer version, Xalan 2.6 for validation. The F-measures of software defect prediction for traditional, DBN, CNN, and improved CNN models were 0.627, 0.681, 0.676, and 0.780, respectively. The table highlights the projects in which our improved CNN model outperformed the other methods. On the basis of the average results of all projects, our improved CNN outperformed the state-of-the-art CNN model by 2.2%, which was a minor improvement yet showed that our model was comparable to the CNN model [20]. The CNN model [20] is a simple model that uses only one

convolutional layer and one pooling layer. In comparison, our model used three convolutional layers and pooling layers, as well as dropouts, l2 regularizations, and He_normal weight initialization. Thus, it was reasonable to assume that our improved model would perform no worse than the CNN model [20].

**Table 4.** Performance comparison of different models on the SPSC dataset. The decimal values represent the average F-measure values. The best F-measure values are highlighted in bold.

| Project | Traditional (LR) | DBN | Li's CNN | Improved CNN |
|---|---|---|---|---|
| Camel | 0.329 | 0.335 | **0.505** | 0.487 |
| JEdit | 0.573 | 0.480 | **0.631** | 0.590 |
| Lucene | 0.618 | 0.758 | **0.761** | 0.701 |
| Xalan | 0.627 | 0.681 | 0.676 | **0.780** |
| Xerces | 0.273 | 0.261 | 0.311 | **0.667** |
| Synapse | 0.500 | 0.503 | 0.512 | **0.655** |
| Poi | 0.748 | **0.780** | 0.778 | 0.444 |
| average | 0.524 | 0.543 | 0.596 | **0.618** |

To check whether our improved model would outperform other models statistically, we first performed the Friedman test. The Friedman test yielded a result of $p = 0.034 < 0.05$, which means that the null hypothesis that all models perform equally was rejected. We then performed the Holm–Bonferroni test, and the results can be seen in Table 5. From Table 5, we found that the performance of the CNN and improved CNN models were significantly different from that of the traditional model, and there were no significant differences between the performance of the CNN and improved CNN models. Considering the average F-measure, we further discovered that the DBN, CNN, and improved CNN models were significantly better performers than the traditional model, and there were no significant differences between the DBN, CNN, and improved CNN models.

**Table 5.** Holm–Bonferroni test of prediction on the SPSC dataset.

| | Traditional (LR) | DBN | Li's CNN |
|---|---|---|---|
| **DBN** | 0.1098 | - | - |
| **Li's CNN** | 0.0038 | 0.0790 | - |
| **Improved CNN** | 0.0079 | 0.1563 | 0.5339 |

However, we could not conclude that our model could outperform the state-of-the-art machine learning models for WPDP. First, due to the limited size of SPSC dataset, experiments performed on the SPSC dataset would not give convincing results. There are only 12 versions of six projects in the SPSC dataset, and these versions could not represent the PROMISE repository very well. Second, the baseline selection of traditional methods is not convincing. Only the logistic regression model is selected as the baseline model which uses traditional hand-crafted features. There is no evidence that the logistic regression model could get better performance than other machine learning models, such as naïve Bayes, random forest, and decision tree.

Summary: Our improved CNN model outperformed the traditional baseline for cross-version WPDP, and the improved CNN model was comparable to the DBN and CNN model, which proved the validity of our model. Although our improved model was significantly better than the traditional model, we could not conclude that our model could outperform the state-of-the-art machine learning models for WPDP due to the limited dataset size and inappropriate baseline selection.

**RQ1b: Does our improved CNN model outperform the state-of-the-art machine learning models on the PSC dataset for within-version WPDP?**

Since performance results on the SPSC dataset could not represent the state-of-the-art machine learning models because of limited dataset size and inappropriate selection of traditional baseline, we further explored the ability of our improved model for WPDP on larger datasets, for example, the PSC dataset. We compared our results with five machine learning models, FIX, and RANDOM baseline. We followed the within-version WPDP pattern for the experiment.

We present our experimental results in terms of F-measure, G-measure, and MCC in Tables 6–8. From the three tables, we observed that FIX got 0 for each version in terms of G-measure and MCC, and RANDOM got results close to 0.5 and 0 in terms of G-measure and MCC. Both models hardly even got the best scores for any version of a project, which indicated that the improved CNN model and other five machine learning models performed well. We also observed that our improved CNN model ranked first for F-measure, G-measure, and MCC. Most significantly, our model improved the F-measure by 6% as compared with the DT model. As for G-measure and MCC, our model improved on the RF and NB models by 5% and 2%, respectively. The results indicated that our improved CNN model beat state-of-the-art machine learning models in terms of F-measure, G-measure, and MCC metrics.

To check whether our improved model outperformed other models statistically, we first performed the Friedman test. The Friedman test yielded a result of $p = 0.00042$, $p = 0.000035$, and $p = 0.005668$ for F-measure, G-measure and MCC, respectively, and they were all below the threshold 0.05, which means that the null hypothesis that all models perform equally was rejected. We then performed the Holm–Bonferroni test, and the results can be seen in Tables 9–11. As the tables indicate, the improved CNN was significantly better than the other five models in terms of F-measure and G-measure. As for MCC, the improved CNN and RF models were significantly better than other models, and there were no significant differences between the CNN and RF models.

**Table 6.** Performance comparison of different models on the PSC dataset. The decimal values represent the average F-measure values. The best F-measures for each row are highlighted in bold.

| Project. | Version | DT | RF | LR | NB | NET | FIX | RANDOM | Improved CNN |
|---|---|---|---|---|---|---|---|---|---|
| Ant | 1.3 | 0.36 | 0.31 | 0.36 | 0.43 | 0.14 | 0.28 | 0.24 | **0.67** |
| | 1.4 | 0.40 | 0.24 | 0.22 | **0.44** | 0.04 | 0.37 | 0.32 | 0.38 |
| | 1.5 | **0.42** | 0.37 | 0.41 | 0.35 | 0.15 | 0.20 | 0.18 | 0.25 |
| | 1.6 | 0.55 | **0.59** | 0.57 | 0.58 | **0.59** | 0.42 | 0.33 | 0.41 |
| | 1.7 | 0.53 | 0.52 | 0.50 | **0.56** | 0.45 | 0.36 | 0.31 | 0.39 |
| Camel | 1.0 | 0.00 | 0.00 | 0.00 | 0.30 | 0.00 | 0.07 | 0.08 | **0.40** |
| | 1.2 | 0.46 | 0.40 | 0.35 | 0.32 | 0.26 | 0.52 | 0.41 | **0.69** |
| | 1.4 | 0.30 | 0.27 | 0.18 | 0.26 | 0.05 | 0.29 | 0.25 | **0.46** |
| | 1.6 | 0.27 | 0.25 | 0.18 | 0.30 | 0.16 | 0.33 | 0.28 | **0.52** |
| Ivy | 1.1 | 0.73 | 0.71 | 0.71 | 0.54 | 0.73 | 0.72 | 0.55 | **0.80** |
| | 1.4 | 0.00 | 0.00 | 0.15 | 0.17 | 0.11 | 0.12 | 0.11 | **0.22** |
| | 2.0 | 0.17 | 0.31 | 0.31 | **0.40** | 0.16 | 0.20 | 0.18 | 0.31 |
| JEdit | 3.2 | 0.56 | 0.63 | 0.68 | 0.55 | 0.59 | 0.50 | 0.40 | **0.69** |
| | 4.0 | 0.49 | **0.56** | 0.49 | 0.41 | 0.43 | 0.39 | 0.31 | 0.48 |
| | 4.1 | 0.45 | 0.54 | **0.61** | 0.52 | 0.54 | 0.40 | 0.33 | 0.41 |
| | 4.2 | 0.46 | 0.29 | 0.37 | 0.44 | 0.38 | 0.23 | 0.20 | **0.58** |
| | 4.3 | 0.00 | 0.00 | 0.00 | **0.21** | 0.00 | 0.04 | 0.04 | 0.00 |
| Log4j | 1.0 | 0.55 | 0.53 | 0.55 | 0.61 | 0.59 | 0.40 | 0.36 | **0.77** |
| | 1.1 | 0.66 | 0.73 | 0.63 | 0.72 | **0.76** | 0.51 | 0.43 | 0.40 |
| | 1.2 | 0.95 | 0.96 | 0.94 | 0.67 | 0.96 | 0.96 | 0.63 | **0.97** |
| Lucene | 2.0 | 0.58 | 0.59 | 0.65 | 0.54 | 0.66 | 0.64 | 0.48 | **0.74** |
| | 2.2 | 0.62 | 0.67 | 0.70 | 0.48 | 0.66 | **0.74** | 0.53 | 0.63 |
| | 2.4 | 0.72 | 0.73 | 0.74 | 0.54 | 0.71 | 0.75 | 0.54 | **0.77** |
| Pbeans | 1.0 | 0.87 | **0.90** | 0.79 | 0.81 | 0.81 | 0.87 | 0.66 | 0.89 |
| | 2.0 | 0.22 | 0.35 | 0.48 | 0.25 | 0.14 | 0.33 | 0.30 | **0.67** |
| Poi | 1.5 | **0.79** | 0.75 | 0.73 | 0.46 | 0.77 | 0.75 | 0.55 | 0.61 |
| | 2.0 | 0.26 | 0.25 | 0.23 | **0.27** | 0.14 | 0.21 | 0.20 | 0.13 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 2.5 | 0.82 | 0.85 | 0.82 | 0.56 | 0.82 | 0.78 | 0.58 | **0.90** |
| | 3.0 | 0.83 | **0.84** | 0.81 | 0.49 | 0.82 | 0.78 | 0.56 | 0.76 |
| Synapse | 1.0 | 0.08 | 0.09 | 0.32 | **0.41** | 0.19 | 0.18 | 0.16 | 0.29 |
| | 1.1 | 0.53 | 0.51 | 0.58 | 0.56 | 0.52 | 0.43 | 0.35 | **0.67** |
| | 1.2 | 0.60 | 0.59 | 0.55 | 0.59 | 0.56 | 0.50 | 0.39 | **0.69** |
| Velocity | 1.4 | 0.90 | 0.89 | 0.89 | 0.89 | **0.91** | 0.86 | 0.60 | 0.90 |
| | 1.5 | 0.80 | **0.83** | 0.80 | 0.45 | 0.78 | 0.80 | 0.57 | 0.78 |
| | 1.6 | 0.54 | 0.50 | 0.51 | 0.38 | 0.47 | 0.51 | 0.41 | **0.83** |
| Xalan | 2.4 | 0.26 | 0.28 | 0.28 | **0.37** | 0.16 | 0.26 | 0.23 | 0.25 |
| | 2.5 | 0.67 | 0.65 | 0.58 | 0.38 | 0.48 | 0.65 | 0.49 | **0.70** |
| | 2.6 | 0.72 | 0.73 | 0.69 | 0.61 | 0.62 | 0.63 | 0.47 | **0.76** |
| Xerces | Initial | 0.71 | 0.68 | **0.73** | 0.34 | 0.63 | 0.64 | 0.48 | 0.65 |
| | 1.2 | 0.38 | 0.33 | 0.07 | 0.23 | 0.03 | 0.28 | 0.24 | **0.41** |
| | 1.3 | 0.50 | 0.44 | 0.42 | 0.38 | 0.35 | 0.26 | 0.22 | **0.56** |
| average | | 0.51 | 0.50 | 0.50 | 0.46 | 0.45 | 0.47 | 0.36 | **0.57** |

**Table 7.** Performance comparison of different models on the PSC dataset. The decimal values represent the average G-measure values. The best G-measures for each row are highlighted in bold.

| Project | Version | DT | RF | LR | NB | NET | FIX | RANDOM | Improved CNN |
|---|---|---|---|---|---|---|---|---|---|
| Ant | 1.3 | 0.50 | 0.39 | 0.50 | 0.63 | 0.18 | 0.00 | 0.49 | **0.67** |
| | 1.4 | 0.56 | 0.29 | 0.29 | **0.58** | 0.05 | 0.00 | 0.51 | 0.51 |
| | 1.5 | **0.51** | 0.44 | 0.47 | 0.69 | 0.17 | 0.00 | 0.50 | 0.28 |
| | 1.6 | 0.66 | 0.66 | 0.64 | 0.67 | **0.69** | 0.00 | 0.48 | 0.53 |
| | 1.7 | 0.65 | 0.60 | 0.55 | **0.68** | 0.50 | 0.00 | 0.51 | 0.46 |
| Camel | 1.0 | 0.00 | 0.00 | 0.00 | **0.62** | 0.00 | 0.00 | 0.54 | 0.50 |
| | 1.2 | 0.54 | 0.46 | 0.41 | 0.36 | 0.28 | 0.00 | 0.49 | **0.76** |
| | 1.4 | 0.37 | 0.30 | 0.20 | 0.35 | 0.05 | 0.00 | 0.49 | **0.54** |
| | 1.6 | 0.35 | 0.29 | 0.19 | 0.37 | 0.17 | 0.00 | 0.50 | **0.60** |
| Ivy | 1.1 | 0.62 | 0.67 | 0.67 | 0.55 | **0.69** | 0.00 | 0.51 | 0.65 |
| | 1.4 | 0.00 | 0.00 | 0.22 | 0.31 | 0.12 | 0.00 | 0.48 | **0.49** |
| | 2.0 | 0.18 | 0.37 | 0.37 | **0.60** | 0.18 | 0.00 | 0.49 | 0.40 |
| JEdit | 3.2 | 0.65 | 0.69 | **0.74** | 0.61 | 0.64 | 0.00 | 0.50 | 0.73 |
| | 4.0 | 0.59 | 0.63 | 0.56 | 0.49 | 0.48 | 0.00 | 0.48 | **0.65** |
| | 4.1 | 0.55 | 0.60 | **0.68** | 0.59 | 0.58 | 0.00 | 0.49 | 0.00 |
| | 4.2 | 0.60 | 0.34 | 0.42 | 0.58 | 0.42 | 0.00 | 0.49 | **0.78** |
| | 4.3 | 0.00 | 0.00 | 0.00 | **0.53** | 0.00 | 0.00 | 0.49 | 0.00 |
| Log4j | 1.0 | 0.66 | 0.60 | 0.64 | 0.66 | 0.65 | 0.00 | 0.52 | **0.81** |
| | 1.1 | 0.72 | 0.76 | 0.70 | 0.77 | **0.79** | 0.00 | 0.52 | 0.44 |
| | 1.2 | 0.40 | 0.31 | 0.12 | **0.63** | 0.00 | 0.00 | 0.45 | 0.00 |
| Lucene | 2.0 | 0.62 | 0.63 | 0.68 | 0.58 | 0.69 | 0.00 | 0.51 | **0.75** |
| | 2.2 | 0.53 | **0.62** | 0.56 | 0.49 | 0.52 | 0.00 | 0.49 | **0.62** |
| | 2.4 | 0.67 | 0.67 | **0.69** | 0.55 | 0.65 | 0.00 | 0.49 | 0.00 |
| Pbeans | 1.0 | 0.75 | 0.77 | 0.60 | 0.71 | 0.28 | 0.00 | 0.53 | **0.89** |
| | 2.0 | 0.32 | 0.45 | **0.68** | 0.33 | 0.18 | 0.00 | 0.50 | 0.67 |
| Poi | 1.5 | **0.74** | 0.69 | 0.61 | 0.47 | 0.61 | 0.00 | 0.50 | 0.61 |
| | 2.0 | 0.32 | 0.32 | 0.28 | 0.39 | 0.15 | 0.00 | **0.51** | 0.25 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 2.5 | 0.76 | 0.81 | 0.72 | 0.56 | 0.71 | 0.00 | 0.52 | **0.82** |
| | 3.0 | 0.75 | **0.78** | 0.73 | 0.49 | 0.72 | 0.00 | 0.50 | 0.72 |
| Synapse | 1.0 | 0.12 | 0.12 | 0.47 | **0.74** | 0.22 | 0.00 | 0.47 | 0.49 |
| | 1.1 | 0.63 | 0.57 | 0.66 | 0.68 | 0.56 | 0.00 | 0.49 | **0.70** |
| | 1.2 | 0.67 | 0.65 | 0.62 | 0.66 | 0.62 | 0.00 | 0.49 | **0.72** |
| Velocity | 1.4 | 0.71 | **0.72** | 0.70 | 0.70 | 0.67 | 0.00 | 0.50 | 0.66 |
| | 1.5 | 0.60 | **0.73** | 0.65 | 0.46 | 0.57 | 0.00 | 0.51 | 0.72 |
| | 1.6 | 0.62 | 0.58 | 0.58 | 0.42 | 0.51 | 0.00 | 0.51 | **0.84** |
| Xalan | 2.4 | 0.34 | 0.33 | 0.32 | **0.51** | 0.18 | 0.00 | 0.50 | 0.31 |
| | 2.5 | 0.66 | 0.67 | 0.61 | 0.41 | 0.53 | 0.00 | 0.50 | **0.69** |
| | 2.6 | 0.74 | 0.75 | 0.71 | 0.62 | 0.65 | 0.00 | 0.49 | **0.78** |
| Xerces | Initial | **0.73** | 0.70 | 0.49 | 0.36 | 0.41 | 0.00 | 0.49 | 0.58 |
| | 1.2 | 0.49 | 0.38 | **0.73** | 0.32 | 0.65 | 0.00 | 0.49 | 0.66 |
| | 1.3 | 0.60 | 0.51 | 0.08 | 0.50 | 0.03 | 0.00 | 0.48 | **0.65** |
| average | | 0.52 | 0.51 | 0.50 | 0.54 | 0.41 | 0.00 | 0.50 | **0.56** |

**Table 8.** Performance comparison of different models on the PSC dataset. The decimal values represent the average MCC measure values. The best MCC measures for each row are highlighted in bold.

| Project | Version | DT | RF | LR | NB | NET | FIX | RANDOM | Improved CNN |
|---|---|---|---|---|---|---|---|---|---|
| Ant | 1.3 | 0.24 | 0.23 | 0.24 | 0.30 | 0.05 | 0.00 | 0.00 | **0.68** |
| | 1.4 | 0.22 | 0.12 | 0.06 | **0.23** | –0.04 | 0.00 | 0.02 | 0.20 |
| | 1.5 | **0.38** | 0.33 | **0.38** | 0.27 | 0.14 | 0.00 | 0.00 | 0.24 |
| | 1.6 | 0.40 | **0.49** | 0.45 | 0.45 | 0.46 | 0.00 | 0.03 | 0.22 |
| | 1.7 | 0.40 | 0.43 | 0.42 | **0.44** | 0.39 | 0.00 | 0.01 | 0.30 |
| Camel | 1.0 | 0.00 | -0.02 | –0.02 | 0.28 | –0.01 | 0.00 | 0.03 | **0.39** |
| | 1.2 | 0.21 | 0.19 | 0.18 | 0.14 | 0.19 | 0.00 | 0.01 | **0.50** |
| | 1.4 | 0.21 | 0.23 | 0.16 | 0.14 | 0.06 | 0.00 | 0.01 | **0.39** |
| | 1.6 | 0.16 | 0.19 | 0.16 | 0.21 | 0.18 | 0.00 | 0.01 | **0.42** |
| Ivy | 1.1 | 0.31 | 0.34 | 0.34 | 0.26 | 0.38 | 0.00 | 0.04 | **0.48** |
| | 1.4 | -0.02 | 0.00 | 0.11 | 0.11 | **0.16** | 0.00 | 0.01 | **0.16** |
| | 2.0 | 0.23 | 0.27 | 0.27 | **0.31** | 0.17 | 0.00 | 0.01 | 0.25 |
| JEdit | 3.2 | 0.37 | 0.48 | 0.53 | 0.40 | 0.46 | 0.00 | 0.00 | **0.56** |
| | 4.0 | 0.34 | **0.45** | 0.39 | 0.29 | 0.36 | 0.00 | 0.03 | 0.29 |
| | 4.1 | 0.30 | 0.45 | **0.50** | 0.40 | 0.46 | 0.00 | 0.02 | 0.00 |
| | 4.2 | 0.39 | 0.26 | 0.33 | 0.36 | 0.37 | 0.00 | 0.01 | **0.51** |
| | 4.3 | 0.00 | -0.01 | –0.02 | **0.20** | 0.00 | 0.00 | 0.00 | 0.00 |
| Log4j | 1.0 | 0.40 | 0.42 | 0.42 | 0.53 | 0.49 | 0.00 | 0.04 | **0.69** |
| | 1.1 | 0.50 | 0.62 | 0.46 | 0.60 | **0.66** | 0.00 | 0.04 | 0.29 |
| | 1.2 | **0.27** | **0.27** | 0.04 | 0.17 | –0.02 | 0.00 | 0.04 | 0.00 |
| Lucene | 2.0 | 0.24 | 0.30 | 0.38 | 0.32 | 0.40 | 0.00 | 0.02 | **0.53** |
| | 2.2 | 0.07 | 0.23 | 0.21 | 0.19 | 0.12 | 0.00 | 0.01 | **0.29** |
| | 2.4 | 0.34 | 0.34 | **0.37** | 0.33 | 0.30 | 0.00 | 0.01 | 0.00 |
| Pbeans | 1.0 | 0.49 | 0.57 | 0.23 | 0.37 | 0.02 | 0.00 | 0.09 | **0.63** |
| | 2.0 | 0.06 | 0.23 | 0.33 | 0.13 | 0.04 | 0.00 | 0.05 | **0.67** |
| Poi | 1.5 | **0.48** | 0.39 | 0.28 | 0.24 | 0.35 | 0.00 | 0.00 | 0.29 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 2.0 | **0.22** | 0.19 | 0.18 | 0.19 | 0.15 | 0.00 | 0.01 | 0.01 |
| | 2.5 | 0.52 | 0.60 | 0.47 | 0.28 | 0.46 | 0.00 | 0.03 | **0.70** |
| | 3.0 | 0.52 | **0.56** | 0.49 | 0.26 | 0.48 | 0.00 | 0.00 | 0.42 |
| Synapse | 1.0 | 0.01 | 0.04 | 0.25 | **0.35** | 0.18 | 0.00 | 0.03 | 0.20 |
| | 1.1 | 0.36 | 0.39 | 0.46 | 0.38 | 0.43 | 0.00 | 0.01 | **0.60** |
| | 1.2 | 0.41 | 0.44 | 0.39 | 0.42 | 0.41 | 0.00 | 0.01 | **0.59** |
| Velocity | 1.4 | 0.56 | 0.52 | 0.51 | 0.53 | **0.59** | 0.00 | 0.01 | 0.56 |
| | 1.5 | 0.33 | **0.49** | 0.38 | 0.19 | 0.27 | 0.00 | 0.02 | 0.42 |
| | 1.6 | 0.34 | 0.31 | 0.32 | 0.24 | 0.33 | 0.00 | 0.02 | **0.76** |
| Xalan | 2.4 | 0.17 | 0.23 | 0.26 | **0.27** | 0.16 | 0.00 | 0.00 | 0.18 |
| | 2.5 | 0.32 | 0.37 | 0.25 | 0.13 | 0.16 | 0.00 | 0.01 | **0.38** |
| | 2.6 | 0.48 | 0.53 | 0.47 | 0.45 | 0.40 | 0.00 | 0.01 | **0.58** |
| Xerces | Initial | **0.47** | 0.42 | 0.46 | 0.19 | 0.32 | 0.00 | 0.01 | 0.23 |
| | 1.2 | **0.29** | 0.28 | 0.04 | 0.12 | 0.11 | 0.00 | 0.01 | 0.28 |
| | 1.3 | 0.43 | 0.40 | 0.37 | 0.29 | 0.31 | 0.00 | 0.02 | **0.50** |
| average | | 0.30 | 0.33 | 0.30 | 0.29 | 0.27 | 0.00 | 0.02 | **0.38** |

**Table 9.** Holm–Bonferroni test of prediction on the PSC dataset for F-measures.

| | **DT** | **RF** | **LR** | **NB** | **NET** |
|---|---|---|---|---|---|
| **RF** | 1 | - | - | - | - |
| **LR** | 0.0857 | 0.0857 | - | - | - |
| **NB** | 0 | 0 | 0.0752 | - | - |
| **NET** | 0 | 0 | 0 | 0.0605 | - |
| **Improved CNN** | 0 | 0 | 0 | 0 | 0 |

**Table 10.** Holm–Bonferroni test of prediction on the PSC dataset for G-measures.

| | **DT** | **RF** | **LR** | **NB** | **NET** |
|---|---|---|---|---|---|
| **RF** | 1 | - | - | - | - |
| **LR** | 0.0231 | 0.0869 | - | - | - |
| **NB** | 0.0559 | 0.1766 | 1 | - | - |
| **NET** | 0 | 0 | 0 | 0 | - |
| **Improved CNN** | 0.0231 | 0.0028 | 0 | 0 | 0 |

**Table 11.** Holm–Bonferroni test of prediction on the PSC dataset for MCCs.

| | **DT** | **RF** | **LR** | **NB** | **NET** |
|---|---|---|---|---|---|
| **RF** | 0.0069 | - | - | - | - |
| **LR** | 0.5251 | 0.0009 | - | - | - |
| **NB** | 0.2749 | 0 | 0.4681 | - | - |
| **NET** | 0.0026 | 0 | 0.0169 | 0.2749 | - |
| **Improved CNN** | 0 | 0.1102 | 0 | 0 | 0 |

After further observation of results for each version of projects, we found that our model performed poorly on JEdit 4.3 in terms of F-measure and G-measure. This was because our model got TP = 0, which means that our model could not find buggy files on JEdit 4.3. Considering that the buggy rate of JEdit 4.3 is 2.3%, we can conclude that our model is weak at predicting ultra-low buggy rate versions of projects, and this is partly acceptable because other baseline models also performed poorly. Another finding is that our model performed poorly on Log4j 1.2 and Lucene 2.4. This was because our model got TN = 0, which means that our model could not find clean files on both versions. Considering that the buggy rate of Log4j 1.2 and Lucene 2.4 is 95.9% and 61.5%, we can conclude that

our model is not robust when predicting ultra-high buggy rate version of projects in terms of G-measure. In conclusion, our model performed not as well towards extreme buggy rate conditions.

Summary: Our improved CNN model outperformed the state-of-the-art machine learning models for within-version WPDP in terms of F-measure, G-measure, and MCC. Moreover, statistical hypothesis tests showed that the improved model was significantly better than the state-of-the-art machine learning models in terms of F-measure and G-measure, and our model could perform as well as RF in terms of MCC.

### *5.3. Performance Under Different Hyperparameter Settings*

**RQ2: How does the improved CNN model perform under different hyperparameter settings?**

As a CNN-based deep model, our model featured various hyperparameters that required tuning. We used the SPSC dataset from the PROMISE repository to tune the hyperparameters. After initial exploration, we fixed some of the hyperparameters as follows: We set batch size to 256, which fit our RAM; the number of epochs to 50 using early stopping strategy; the learning rate to 0.0015 after initial selections; the convolutional layers and pooling layers to 3; the regularization alpha to 0.0001 to reduce generalization error; the dropout to 0.5 as suggested in [36]; the filter length to 2; the numbers of filters to 10; and the embedding output to 32. These hyperparameters applied to our experiment on the PSC and SPSC dataset. The remaining hyperparameters included dense layers and dense nodes as well as kernel size and stride.

Figure 9 shows the F-measure of our improved CNN model under different numbers of dense layers and dense nodes. We observed that our model worked best under dense layers = 4 and dense nodes = 50. Adding more dense layers had a significant impact on F1. Dense node numbers were less significant for F-measure as compared with dense layer numbers. When the number of dense layers increased, the number of dense nodes had to decrease accordingly to avoid increasing the model capacity.

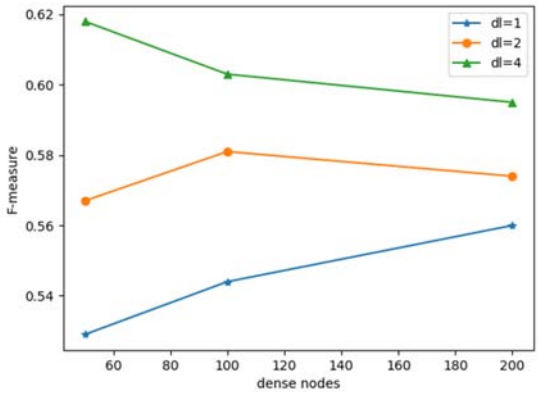


**Figure 9.** Dense layers and dense nodes. Here dl means dense layers.

Figure 10 shows the F-measure of our improved CNN model under different numbers of kernel sizes and strides. The epochs were set to 10 for speed acceleration. We observed that our model worked best under kernel size = 2 and stride = 2, which was slightly better than kernel size = 3 and stride = 3. There was a minor difference between choosing kernel size and stride of 2 or 3. However, kernel size and stride had to be matched in size, or there would have been a shape degradation in performance.

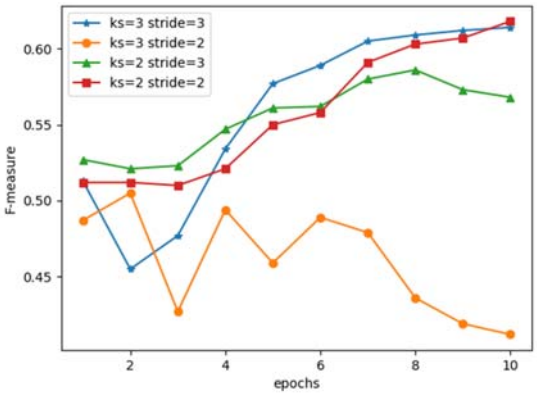


**Figure 10.** Kernel size and stride. Here ks means kernel size.

The tuned hyperparameters were specified for the exact dataset, (i.e., SPSC dataset). However, when using this model for other projects, we can fine tune these hyperparameters based on the characteristic of the projects. For example, if we were to utilize the model to predict projects with larger source files, the length of the input vector to the embedding layer would be expanded. Thus, some hyperparameters such as embedding output, kernel size, and stride should be adjusted to fit the model capacity.

### *5.4. Hyperparameter Instability of Deep Learning Models*

**RQ3: Could our model generate results comparable to those of other baseline methods for each version of a project?**

To answer the research question, we further analyzed experimental data on the SPSC dataset. Before analyzing, we first defined the concept of hyperparameter instability as described below.

**Hyperparameter instability** indicates that the performance of a model is significantly different from that of the same model under different hyperparameter settings in terms of each version or project, but that it performs similarly on average.

Hyperparameter instability is not limited to deep learning models. However, deep learning models would respond more negatively to hyperparameter instability, because machine learning models inherently have fewer hyperparameters for tuning. Thus, it is relatively difficult to get two machine learning models that feature different hyperparameter settings, however, get similar average scores. As for deep learning models, there are so many hyperparameters for tuning, for example, learning rate, network layers, hidden nodes in each layer, activation functions, and training optimizers. Given the fact that, usually, not all hyperparameters are provided in detail in a paper, researchers who aim to replicate the model could build a model of different hyperparameters, and the model would be more likely to respond negatively to hyperparameter instability.

Supporting evidence was provided by the experimental results on the SPSC dataset. Although our improved model was not significantly different from that of the CNN model, as Table 4 supports, we discovered that our improved model performed dramatically better for Xalan, Xerces, and Synapse, for which it improved the F-measure by 10.4%, 35.6%, and 14.3%, respectively. However, our model performed dramatically worse for Lucene and Poi, decreasing the F-measure by 6% and 33.4%, respectively. When focused on Poi, our model performed dramatically worse as compared with traditional methods.

The analysis above shows that our improved model responded negatively to hyperparameter instability. While the model could get a high average score, it was less robust, and it tended to fluctuate when focused on an individual version or project. A reasonable explanation for this phenomenon is that the combination of AST-based semantic features and deep learning models required deep learning models to possess a larger model capacity as compared with machine learning models, which may have added to the instability of the deep models. Thinking more positively,

however, the phenomenon may reveal not only that different kinds of classification models could predict different kinds of defects, but also that different hyperparameter settings of the same deep learning model could predict different kinds of defects as well. If the phenomenon of hyperparameter instability is verified in future research, deep ensemble models may be used to further enhance defect prediction.

## 6. Threats to Validity

Deletion of files when parsing source code. When we utilized javalang to parse the source files, it did not always build the parse trees successfully due to its limited syntax support of the Java programming language. We used a simple strategy to delete the source files that could not be parsed by javalang. The statistical results showed that we removed 3.7% of source files on SPSC dataset and 2.1% on PSC dataset. As for average buggy rate, we increased it by 0.6% as compared with the original PROMISE repository, which shows that we did not simply delete buggy files or clean files, but we deleted files in a rather balanced way. Therefore, we claim that deletion of files would not influence the validity of our results that much.

Quality of the PSC dataset. We designed the PSC dataset to target AST-based features, which enlarged the existing SPSC dataset from 14 to 41 versions. We also took measures to make sure that the link between open source project versions and labeled CSV files were verified and that several evaluation metrics could be correctly computed. However, we cannot guarantee similar experimental results on projects outside of the PSC dataset.

Programming language. Our dataset came from the PROMISE repository, where projects are written in the Java programming language. Although our model and preprocessing strategy could be applied to other programming languages (i.e., C, python), we cannot guarantee similar experimental results.

Replication of baseline models. We did not replicate our baseline models. Instead, we made our best efforts to conform to experimental environments listed in related papers, and we compared our experimental results with those listed in related papers. Because some detailed information is not given in those papers, it may influence experiment validity.

## 7. Related Works

### *7.1. Deep Learning Based Software Defect Prediction*

Since 2012, deep learning has been used in various domains including software engineering. In software defect prediction, deep learning first emerges in 2015, and it is used more frequently since then. Up to now, many researchers have explored the use of deep learning in software defect prediction. According to the feature type of deep learning-based software defect prediction, it is further divided into two categories:

1. Defect prediction based on hand-crafted features

When using various kinds of traditional hand-crafted features, how to combine existing features to generate more effective features remains a problem. Deep learning models own the ability of effective feature combination. Therefore, deep learning models could be used in such situations to enhance model performance. Yang et al. [50] proposed a deep learning model for just-in-time defect prediction, which predicts defect-prone changes. They selected 14 basic change measures regarding code change, and leveraged DBN model to build a set of expressive features from these basic measures. At last, they used machine learning models for classification. Experiments show that their methods could discover 32.22% more bugs than the state-of-the-art model. Tong et al. [51] proposed a deep learning model and two-stage ensemble learning for defect prediction. They leveraged stacked denoising autoencoders to generate effective features from traditional hand-crafted features in the NASA MDP dataset, and used ensemble classifiers for defect prediction. The results showed that deep representations of existing metrics are promising for software defect prediction.

2. Defect prediction based on deep features

Deep feature-based software defect prediction does not use hand-crafted features. Instead, this kind of method generates deep features from source codes or ASTs. In 2016, Wang et al. [23] leveraged DBN for software defect prediction. They used selected AST sequences taken from source codes as input to the DBN model, which generate new expressive features, and used machine learning models for classification. Their WPDP and cross-version defect prediction experiments showed that their model outperformed the state-of-the-art machine learning models. Then in 2017, Li et al. [24] proposed a CNN-based defect prediction model, which leveraged word embedding and a CNN model for defect prediction. Although they used logistic regression instead of various machine learning models for classification, their results outperformed the DBN models [23]. They also proved that adding traditional features to deep features could further enhance model performance. In 2018, two papers leveraging recurrent neural network (RNN) were published. The first paper [52] used a type of RNN model, long-short term memory (LSTM) model, to predict defects, which takes AST sequences as the input. The second paper [53] leveraged tree-based LSTM models to predict defects, which takes AST as the input. However, their results were not as good as results for Li's model [24]. There is also research on deep defect prediction targeting assembly code [54,55], both of which leveraged a CNN model to learn from assembly instructions.

### *7.2. Deep Learning in Software Engineering*

Apart from software defect prediction, deep learning is also used in various software engineering domains. Generally, deep learning models are used in software maintenance [56], code clone detection [57], defect detection and localization [58,59], and other domains.

In software maintenance, Guo et al. [56] used a RNN model to establish links between requirements, design, source code, test cases, and other artifacts, which is called trace links. Their results outperformed the state-of-the-art tracing methods including the vector space model and the latent semantic indexing. In code clone detection, Li et al. [57] proposed a deep learning-based clone detection approach. Code clone refers to copied code with or without modification, which could challenge software maintenance. They used AST tokens to represent method-level code clones and nonclones to train a classifier, and then used the classifier to detect code clones. Their methods achieved similar performance with low time cost. In defect detection and localization, Nguyen et al. [58] utilized deep neural network for bug localization. The aim of the model was to solve lexical mismatch problem, which references that the terms used in bug report are different from the terms and code tokens used in source files. Their model achieved 70% accuracy with five suggested files. Pradel and Sen [59] leveraged DeepBugs, which is a learning approach to detect name-based bugs. They used word embedding to form semantic features from methods, and they used learn-bug detectors instead of manually writing them. Their approach achieved high accuracy, high efficiency, and disclosed 102 programming mistakes in real-world code.

Other software engineering domains also leverage deep learning, such as source code classification [60], runtime behavior analysis [61], feature location [62], vulnerability analysis [63], code author identification [64], and so on.

## 8. Conclusions and Future Work

We proposed an improved CNN model which could better learn semantic representations from source-code ASTs for WPDP. On the basis of Li's CNN model, we made further improvements by enhancing global pattern capture ability and improving the model for better generalization.

To verify that the CNN model could outperform state-of-the-art methods, we performed two experiments. The first experiment was performed on the SPSC dataset to demonstrate that our model was comparable to the existing CNN model for WPDP. The results showed that our model improved the existing CNN model by 2.2% in terms of F-measure, and statistical hypothesis tests showed that our method was comparable to the existing DBN and CNN models, and significantly better than the traditional baseline. The second experiment was performed on the PSC dataset, which was designed especially for AST-based features extracted from the source code to validate that our model could outperform the state-of-the-art machine learning models for WPDP. The results showed that our

model improved the F-measure by 6%, the G-measure by 5%, and the MCC by 2% as compared with the best-performing machine learning models among DT, RF, LR, NB, and NET. Statistical hypothesis tests show that our method was almost significantly better than other machine learning models in terms of the evaluation metrics above, except for MCC, for which our model was comparable to the RF model.

On the basis of our experimental results, we also proposed the concept of hyperparameter instability to describe a model whose performance is significantly different from the same model of different hyperparameter settings in terms of each version or project, however, it performs similarly on average. Our improved model responded negatively to hyperparameter instability, which posed a threat to the robustness of the model, however, also suggested that deep ensemble models may enhance defect prediction.

In the future, we would like to collect more C/C++ open source projects and build new datasets for deep-feature-based defect prediction. In addition, it would be promising to use other kinds of deep models such as RNN to generate features for predicting defects automatically. Determining what types of defects could be predicted in deep learning-based defect prediction is also essential. Lastly, ensemble methods of deep learning models for defect prediction could also be a future research direction.

**Author Contributions**: Conceptualization C.P. and B.X.; data curation C.P. and H.G.; methodology C.P.; supervision M.L.; writing – original draft C.P.; revise – original draft B.X. and H.G.; writing—review and editing C.P. and M.L.

**Funding:** This research received no external funding

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

1. Menzies, T.; Milton, Z.; Turhan, B.; Cukic, B.; Jiang, Y.; Bener, A. Defect prediction from static code features: current results, limitations, new approaches. *Autom. Softw. Eng.* **2010**, *17*, 375–407.
2. Moser, R.; Pedrycz, W.; Succi, G. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. *Semantics2017* **2008**, 181.
3. Tan, M.; Tan, L.; Dara, S.; Mayeux, C. Online Defect Prediction for Imbalanced Data. In Proceedings of 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE), Florence, Italy, 16–24 May 2015, pp. 99–108.
4. Nam, J.; Pan, S.J.; Kim, S. Transfer defect learning. In Proceedings of International Conference of Software Engineering, San Francisco, CA, USA, 18–26 May 2013.
5. Nam, J. Survey on software defect prediction, Ph.D. Thesis, The Hong Kong University of Science and Technology, Hong Kong, China, 3 July 2014.
6. Lyu, M.R. In *Handbook of software reliability engineering*, IEEE computer society press: Washington, DC, WA, USA, 1996, vol. 222.
7. Halstead, M.H. In *Elements of Software Science*, Elsevier Science Inc.: New York, NY, USA, 1977.
8. McCabe, T.J. A Complexity Measure. *IEEE Trans. Soft. Eng.* **1976**, *SE-2*, 308–320.
9. Chidamber, S.R.; Kemerer, C.F. A Metrics Suite for Object Oriented Design. *IEEE Trans. Soft. Eng.* **1994**, *20*, 476-493.
10. Harrison, R.; Counsell, S.; Nithi, R. An evaluation of the MOOD set of object-oriented software metrics. *IEEE Trans. Softw. Eng.* **1998**, *24*, 491–496.
11. Jiang, T.; Tan, L.; Kim, S. Personalized defect prediction. In Proceedings of Automated Software Engineering, Silicon Valley, CA, USA, 11–15 November 2013, pp. 279–289.
12. Gyimothy, T.; Ferenc, R.; Siket, I. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Trans. Softw. Eng.* **2005**, *31*, 897–910.
13. Zhou, Y.; Leung, H.K.N. Empirical Analysis of Object-Oriented Design Metrics for Predicting High and Low Severity Faults. *IEEE Trans. Softw. Eng.* **2006**, *32*, 771–789.
14. Lessmann, S.; Baesens, B.; Mues, C.; Pietsch, S. Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings. *IEEE Trans. Softw. Eng.* **2008**, *34*, 485–496.

15. Greenwald, J.; Frank, A.; Menzies, T. Data Mining Static Code Attributes to Learn Defect Predictors. *IEEE Trans. Softw. Eng.* **2007**, *33*, 2–13.
16. Jing, X.Y.; Ying, S.; Wu, S.S.; Liu, J. Dictionary learning based software defect prediction. In Proceedings of the 36th International Conference on Software Engineering, Hyderabad, India, 31 May – 7 June 2014.
17. Hindle, A.; Barr, E.T.; Su, Z.; Gabel, M.; Devanbu, P. On the naturalness of software. In Proceedings of 2012 34th International Conference on Software Engineering (ICSE), Zurich, Switzerland, 2–9 June 2012, pp. 837-847.
18. Nguyen, A.T.; Nguyen, T.N. Graph-based statistical language model for code. In Proceedings of 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Florence, Italy, 16–24 May 2015, pp. 858–868.
19. Shippey, T. Bowes, D.; Hall, T. Automatically identifying code features for software defect prediction: Using AST N-grams. *Inf. Softw. Technol.* **2018**, *106*, 142–160.
20. White, M.; Vendome, C.; Linares-Vasquez, M.; Poshyvanyk, D. Toward Deep Learning Software Repositories. *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories (MSR)* **2015**, 334–345.
21. Xiao, Y.; Keung, J.; Bennin, K.E.; Mi, Q. Machine translation-based bug localization technique for bridging lexical gap. *Inf. Softw. Technol.* **2018**, *99*, 58–61.
22. Tu, Z.; Su, Z.; Devanbu, P. On the localness of software. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Hong Kong, China, 16–21 November 2014, pp. 269–280.
23. Wang, S.; Liu, T.; Tan, L. Automatically learning semantic features for defect prediction. *ICSE '16* **2016**, 297–308.
24. Li, J.; He, P.; Zhu, J.; Lyu, M.R. Software Defect Prediction via Convolutional Neural Network. *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)* **2017**, 318–328.
25. Krizhevsky, A.; Sutskever, I.; Hinton, G.E. Imagenet classification with deep convolutional neural networks. In Proceedings of Advances in Neural Information Processing Systems, Lake Tahoe, NV, USA, 3–8 December 2012, 1097–1105.
26. Goodfellow, I.; Bengio, Y.; Courville, A. Deep learning. *Nature* **2015**, *521*, 436–444.
27. Arcuri, A.; Briand, L. A practical guide for using statistical tests to assess randomized algorithms in software engineering. *ICSE '11* **2011**.
28. Hassan, A.E.; Tantithamthavorn, C.; McIntosh, S.; Matsumoto, K. An Empirical Comparison of Model Validation Techniques for Defect Prediction Models. *IEEE Trans. Softw. Eng.* **2017**, *43*, 1–18.
29. He, Z.; Péters, F.; Menzies, T.; Yang, Y. Learning from Open-Source Projects: An Empirical Study on Defect Prediction. *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* **2013**, 45–54.
30. Jureczko, M.; Madeyski, L. Towards identifying software project clusters with regard to defect prediction. *ICTRS'17* **2010**.
31. Trautsch, A.; Herbold, S.; Grabowski, J. Correction of "A Comparative Study to Benchmark Cross-project Defect Prediction Approaches". *IEEE Trans. Soft. Eng* **2017**, 1–1.
32. Abdel-Hamid, O.; Mohamed, A.-R.; Jiang, H.; Penn, G. Applying Convolutional Neural Networks concepts to hybrid NN-HMM model for speech recognition. *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* **2012**, 4277–4280.
33. LeCun, Y.; Bottou, L.; Bengio, Y. Haffner, P. Gradient-based learning applied to document recognition. *Proc. IEEE* **1998**, *86*, 2278–2324.
34. Zhang, X.; Zhao, J.; LeCun, Y. Character-level Convolutional Networks for Text Classification. In Proceedings of the Proceedings of the Advances in Neural Information Processing Systems; ArXiv, 2015.
35. Thom, M.; Palm G. Sparse activity and sparse connectivity in supervised learning. *J. Mach. Learn. Res.* **2013**, *14*, 1091–1143.
36. Rong, X. word2vec Parameter Learning Explained. Available online: https://arxiv.org/abs/1411.2738 (accessed on 24 January 2019)
37. Srivastava, N.; Hinton, G.; Krizhevsky, A.; Sutskever, I.; Salakhutdinov, R. Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.* **2014**, *15*, 1929–1958.
38. Peng, H.; Mou, L.; Li, G.; Liu, Y.; Zhang, L.; Jin, Z. Building Program Vector Representations for Deep Learning. In Proceedings of the Image Analysis and Processing — ICIAP 2015; Springer Nature, 2015; Vol. 9403, pp. 547–553.

39. He, K.; Zhang, X.; Ren, S.; Sun, J. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *2015 IEEE International Conference on Computer Vision (ICCV)* **2015**, 1026–1034.
40. Glorot, X.; Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. In Proceedings of the thirteenth international conference on artificial intelligence and statistics, Sardinia, Italy, 13–15 May 2010, pp. 249–256.
41. Bottou, L. Large-Scale Machine Learning with Stochastic Gradient Descent. In Proceedings of the Proceedings of COMPSTAT'2010; Springer Nature, 2010; pp. 177–186.
42. Kingma, D.P.; Ba, J. Adam: A method for stochastic optimization. Available online: https://arxiv.org/abs/1412.6980 (accessed on 25 February 2019)
43. D'Ambros, M.; Lanza, M.; Robbes, R. An extensive comparison of bug prediction approaches. *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)* **2010**.
44. Zimmermann, T.; Nagappan, N.; Gall, H.; Giger, E.; Murphy, B. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, Amsterdam, The Netherlands, 24–28 August 2009, pp. 91–100.
45. Shepperd, M.; Song, Q.; Sun, Z.; Mair, C. Data Quality: Some Comments on the NASA Software Defect Datasets. *IEEE Trans. Softw. Eng.* **2013**, *39*, 1208–1215.
46. Herzig, K.; Just, S.; Rau, A.; Zeller, A. Predicting defects using change genealogies. *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)* **2013**, 118–127.
47. Wu, R.; Zhang, H.; Kim, S.; Cheung, S.C. Relink: Recovering links between bugs and changes. In Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, Szeged, Hungary, 5–9 September 2011, pp. 15–25.
48. Friedman, M. The Use of Ranks to Avoid the Assumption of Normality Implicit in the Analysis of Variance. *J. Am. Stat. Assoc.* **1937**, *32*, 675–701.
49. Holm, S. A simple sequentially rejective multiple test procedure. *Scand. J. Stat.* **1979**, *6*, 65–70.
50. Yang, X.; Lo, D.; Xia, X.; Zhang, Y.; Sun, J. Deep Learning for Just-in-Time Defect Prediction. In proceedings of 2015 IEEE International Conference on Software Quality, Reliability and Security (QRS), Vancouver, BC, Canada, 3–5 Augusty 2015, pp. 17–26.
51. Tong, H.; Liu, B.; Wang, S. Software defect prediction using stacked denoising autoencoders and two-stage ensemble learning. *Inf. Softw. Technol.* **2018**, *96*, 94–111.
52. Zhang, X. Using Cross-Entropy Value of Code for Better Defect Prediction. *Int. J. Perform. Eng.* **2018**, *14*, 2105.
53. Dam, H.K.; Pham, T.; Ng, S.W.; Tran, T.; Grundy, J.; Ghose, A.; Kim, C.J. A deep tree-based model for software defect prediction. Available online: https://arxiv.org/abs/1802.00921 (accessed on 24 January 2019)
54. Phan, A.V.; Nguyen, L.M.; Bui, L.T. Convolutional neural networks over control flow graphs for software defect prediction. In proceedings of 2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI), Boston, MA, USA, 6–8 November 2017, pp. 45–52.
55. Phan, A.V.; Nguyen, L.M. Convolutional neural networks on assembly code for predicting software defects. In 2017 21st Asia Pacific Symposium on Intelligent and Evolutionary Systems (IES), Hanoi, Vietnam, 15–17 November 2017.
56. Cheng, J.; Guo, J.; Cleland-Huang, J. Semantically Enhanced Software Traceability Using Deep Learning Techniques. *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)* **2017**, 3–14.
57. Li, L.; Feng, H.; Zhuang, W.; Meng, N.; Ryder, B. CCLearner: A Deep Learning-Based Clone Detection Approach. *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)* **2017**, 249–260.
58. Lam, A.N. Nguyen, A.T.; Nguyen, H.A.; Nguyen, T.N. Bug localization with combination of deep learning and information retrieval. In proceedings of 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), Buenos Aires, Argentina, 22–23 May 2017, pp. 218-229.
59. Pradel, M.; Sen, K. DeepBugs: a learning approach to name-based bug detection. *Proc. ACM Program. Lang.* **2018**, *2*, 1–25.
60. Reyes, J.; Ramirez, D.; Paciello, J. Automatic Classification of Source Code Archives by Programming Language: A Deep Learning Approach. *2016 International Conference on Computational Science and Computational Intelligence (CSCI)* **2016**, 514–519.

61. Zekany, S.; Rings, D.; Harada, N.; Laurenzano, M.A.; Tang, L.; Mars, J. CrystalBall: Statically analyzing runtime behavior via deep sequence learning. *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* **2016**, 1–12.
62. Corley, C.S.; Damevski, K.; Kraft, N.A. Exploring the use of deep learning for feature location. In proceedings of 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), Bremen, Germany, 29 September – 1 October 2015, pp. 556–560.
63. Pang, Y.; Xue, X.; Wang, H. Predicting Vulnerable Software Components through Deep Neural Network. *ICTCE '17* **2017**, 6–10.
64. Bandara, U.; Wijayarathna, G. Deep Neural Networks for Source Code Author Identification. In Proceedings of 20th International Conference, Daegu, Korea, 3–7 November 2013, pp. 368–375.