RESEARCH ARTICLE

Effort-aware and just-in-time defect prediction with neural network

Lei Qiao^{*}, Yan Wang

School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China

* qiaolei@bit.edu.cn

Abstract

Effort-aware just-in-time (JIT) defect prediction is to rank source code changes based on the likelihood of detects as well as the effort to inspect such changes. Accurate defect prediction algorithms help to find more defects with limited effort. To improve the accuracy of defect prediction, in this paper, we propose a deep learning based approach for effort-aware just-in-time defect prediction. The key idea of the proposed approach is that neural network and deep learning could be exploited to select useful features for defect prediction because they have been proved excellent at selecting useful features for classification and regression. First, we preprocess ten numerical metrics of code changes, and then feed them to a neural network whose output indicates how likely the code change under test contains bugs. Second, we compute the benefit cost ratio for each code change by dividing the likelihood by its size. Finally, we rank code changes according to their benefit cost ratio. Evaluation results on a well-known data set suggest that the proposed approach outperforms the state-of-the-art approaches on each of the subject projects. It improves the average recall and popt by 15.6% and 8.1%, respectively.

OPEN ACCESS

Citation: Qiao L, Wang Y (2019) Effort-aware and just-in-time defect prediction with neural network. PLoS ONE 14(2): e0211359. <https://doi.org/10.1371/journal.pone.0211359>

Editor: Jie Zhang, Newcastle University, UNITED KINGDOM

Received: November 8, 2018

Accepted: January 13, 2019

Published: February 1, 2019

Copyright: © 2019 Qiao, Wang. This is an open access article distributed under the terms of the [Creative Commons Attribution License](https://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

Data Availability Statement: The datasets underlying the current study are available at GitHub (<https://github.com/donaldjoe/Effort-Aware-and-Just-in-Time-Defect-Prediction-with-Neural-Network>).

Funding: The author(s) received no specific funding for this work.

Competing interests: The authors have declared that no competing interests exist.

Introduction

Software quality assurance activities, such as the defect prediction and source code inspection, have a great influence on producing high quality reliable software [1, 2]. However, such activities are expensive with the increasing scale of modern software since these software systems are complex and failure-prone. In most cases, the developers have limited resources and insufficient time to test the software [3]. To reduce defects and improve the reliability of such software applications, developers should inspect source code changes thoroughly to identify and fix all defects. However, source code inspection is challenging, labor-intensive, tedious and time consuming. In addition, testing all units for finding bugs is impractical because the software program budgets are finite and the release schedules are tight [4].

To facilitate source code inspection, software defect prediction techniques have been proposed [5]. Defect prediction techniques help software quality assurance team to inspect the code area that detect the most likely defective code [6]. Developers utilize the software defect prediction techniques that automatically detect the potential defects in most likely buggy code area to find the bugs and then allocate the limited resources [7]. Consequently, defect

prediction plays a critical role in software quality assurance activities. Software defect prediction is to predict how likely a given piece of source code may contain defects (or how many defects it may contain) [8–10]. With such defect prediction techniques, developers rank software modules for inspection. Based on such ranks, developers may invest less effort in inspecting those source code changes that are more likely to contain defects. As a result, more defects can be identified with limited inspection effort.

The process of software defect prediction [4, 11, 12] involves three steps. The first step is to collect history data sets for training [13]. To this end, researchers identify and recover defect-inducing changes by building the link between the source code from Concurrent Versions System (CVS) and bug reports from bug tracking system (e.g., bugzilla) to generate such data sets. The second step is to build a classifier which can predict whether a change is buggy or not [11]. The third step is to make predictions based on the resulting classifier [11]. Defect prediction can be applied at different granularity: packages [14], files [15], methods [16, 17] or changes [13, 18]. Change-level defect prediction is proposed by Mockus and Weiss [19] to recommend which changes made to software applications should be inspected first. It is also known as just-in-time (JIT) defect prediction because the prediction are made promptly once source code changes are committed [20]. JIT defect prediction has many advantages. First, JIT defect prediction can narrow down inspecting source code when find defects. Developers can only review the most likely buggy change without inspecting the whole file or package. The amount of code for inspecting in a change is less than in a file or package. Second, JIT defect prediction can help developers identify whether a change is defect-inducing change at check-in time [21], when the code are still fresh in their minds. JIT defect prediction can find the bugs in time and help the developers allocate the limited test resources reasonably in practical applications.

Most defect prediction approaches take defect prediction as a binary classification problem that could be solved by different classification algorithms [22–26], e.g., Support Vector Machine (SVM) [23], Random Forest (RF) [27] and Nearest-Neighbor [28]. Such approaches simply classify source code changes into two categories: buggy or clean. However, such approaches do not take into account the inspection effort required to discover the predicted defects, i.e., the effort of code inspection on suspicious code changes. As a result such approaches maximize classification performance, e.g., precision, recall, and F-measure, but fail to maximize the number of defects identified with given inspection effort (resource) [29]. To this end, new approaches begin to take into account the inspection effort in defect prediction. Such approaches are also called effort-aware defect prediction where effort is usually measured by the number of modified source code within a given code change [30].

To improve software reliability [31] with limited human resource, in this paper we propose an effort-aware just-in-time defect prediction approach based on neural networks and deep learning. Deep learning has been applied to many learning tasks, such as speech recognition, image processing and code completion [32, 33]. We employ neural network and deep learning in defect prediction because they are good at selecting useful input features automatically, and building the complex mapping between input and output [34, 35]. It is suitable for defect prediction since the mapping between the input (metrics of code changes) and the output (number of defects) is very complex, and we do not know exactly which input feature (metrics) are really useful or useless for defect prediction.

We evaluate the proposed approach on a well-known data set where the number of defects in each code change is identified [13]. The results of the ten-fold cross-validation evaluation suggest that the proposed approach outperforms existing approaches. Compared to the well-known CBS (i.e., Classify-Before-Sorting) [36] and LT (i.e., the unsupervised approach of the lines of code in a file before the change) [18], the proposed approach improves the recall [13] and popt [13, 30] by 15.6% and 8.1%, respectively.

This paper makes the following contributions:

- We propose an effort-aware just-in-time defect prediction approach based on neural network.
- We evaluate the proposed approach with a well-known data set and evaluation results suggest that the approach outperforms existing approaches.

The rest of this paper is summarized as follows. Section 2 introduces the background and related work on defect prediction. Section 3 gives the details about our approach, neural network based regression model (denoted as NNR model for short), including: data preprocessing and the methodology of building NNR model. Section 4 provides the evaluation of our research and the results comparison of our approach with previous studies. Section 5 examines the threats to validity of our research. The conclusion is given in Section 6.

Related work

Defect prediction

Software applications are often complex and thus fault-prone. Defect prediction helps developers to locate and fix more bugs quickly with less effort. Akiyama built the first defect prediction model to predict the number of defects based on lines of code (LOC) in 1971 [37]. After that, a number of defect prediction approaches have been proposed [7, 13, 18, 22, 23]. Such approaches can be divided into two categories: supervised approaches and unsupervised approaches. Supervised approaches learn from the history labeled data by training classifiers on such data. To facilitate the research of supervised approaches, Kamei et al. build a data set from six open-source projects by mining source code repositories and bug reports [13]. They propose an effort-aware just-in-time change-level supervised defect prediction approach. They also leverage logistic regression to build a customized supervised logistic regression (LR) model and use the linear regression to build an effort-aware linear regression (EALR) model for the defect prediction. They find that EALR model is better than LR model. Unsupervised defect prediction predicts defect proneness without requiring access to training data. Unsupervised defect prediction approaches, such as clustering techniques, are more appropriate for building the models when there is lacking of software defect data [38]. One main advantage of unsupervised model is that it does not need to collect and label historical defect training data. However, the research of unsupervised defect prediction approaches is very scarce. One reason is that unsupervised defect prediction is usually underperforms supervised ones in terms of prediction performance [39]. Yang et al. [18] present an unsupervised approach that ranks every change in descending order according to the reciprocal of the raw value of every change metric. Surprisingly, they find that the unsupervised model based on LT outperforms the state-of-the-art supervised approach EALR. Fu and Menzies [40] replicate and extend Yang et al.'s [18] study, their experiment results show that not all unsupervised models work better than supervised models for effort-aware just-in-time defect prediction.

Defect prediction can be performed on different granularity. Some defect prediction are made at package level [41]. Package-level defect prediction is based on the hypothesis that the likelihood of a package to fail is significantly determined by the problem domain [14]. The package-level defect prediction is to predict failure-prone components on the package-level (e.g., Martin's metrics) defect history data sets. For example, Mishra et al. [41] propose a new Support Vector based Fuzzy Classification System (SVFCS) for defective module prediction at package level. Some defect prediction are made at method level [16]. Method-level defect prediction is to predict bug-prone methods on the method-level (the McCabe's and Halstead's

ones) data sets. For example, Giger et al. [16] investigate bug prediction models at the level of individual methods. They decrease the granularity of defect prediction and thus reduce manual inspection efforts. In addition, the change level defect prediction has been proposed as well [13, 18]. It makes prediction using change metrics at check-in time [21] when the code impression is still fresh in developers' minds. It is performed whenever software changes are submitted to the version control systems. Mouckus and Weiss [19] first propose just-in-time (JIT) defect prediction and they use all the properties of the software changes to build a logistic model for defect prediction. Kamei et al.'s EALR model [13] and Yang et al.'s LT model [18] perform just-in-time defect prediction on change-level as well. Yang et al. apply the deep learning to the JIT defect prediction [42]. They exploit the Deep Belief Network (DBN) that consists of three Restricted Boltzmann's Machine (RBM) and a logistic regression classifier to build the deeper model. Then, they use the model to detect more expressive features instead of the initial features (i.e., fourteen metrics). Finally, a classifier is built based on the selected expressive features. Given a new change, the classifier can predict the change as buggy or clean based on the confidence scores. In our paper, we use three full connection Back Propagation (BP) neural networks [43] to build a regression model, and not a classifier model. Our neural network regression (NNR) approach exploits 10 change metrics except the number of modified directories (ND), recent developer experience (REXP), the lines of code added (LA) and the lines of code deleted (LD). The output of our neural networks regression approach is a prediction value (i.e., defect probability). Our changes ranking approach is different from Yang et al. We sort our data set according to the predicted defect probability divided by the sum of LA and LD in descending order. Recently, Huang et al. [36] propose the CBS (classify before sorting) approach and conduct defect prediction on change level as well. The core idea of CBS approach is that among the changes classified to be defective, the smaller ones should be inspected first. First, they build a linear classifier following Kamei by using the linear regression. Second, changes are classified into two groups: defective and non-defective by the classifier. Third, They sort the predicted defective changes in ascending order according to their size (i.e., $la + ld$) to build the CBS model. Current defect prediction techniques mainly depend on historical data. However, historical data from past projects may not be available. Consequently, the defect prediction would be difficult. To address this problem, Li et al. propose sample-based methods for building cost effective defect prediction model while keeping the sample size small [44]. They randomly sample a small percentage of modules from a local project to construct a classification model. Then they use the model to predict defect-proneness of the un-sampled modules in the current project, independent from the past project. Their methods involve three methods, sampling with conventional machine learners, sampling with semi-supervised learning (i.e., the CoForest method) and sampling with active semi-supervised learning (i.e., the ACoForest method). Their approach does not require historical data and the defect prediction is made at package-level and file-level. In contrast, our proposed defect prediction approach requires historical data and is made at change-level.

Effort-aware defect prediction and performance indicators

The previous studies on defect prediction do not consider the effort of source code inspection [9, 15, 45]. Most studies of defect prediction simply take the defect prediction as a classification problem, and evaluate the performance with precision, recall and F-measure [23]. In other words, traditional defect prediction approaches assume that the effort to test and inspect software modules is constant regardless of the different sizes of the modules [45]. To take the inspection effort into consideration, Mende and Koschke [30] propose an effort-aware defect prediction model that ranks source code based on not only software defect-proneness (i.e., the

number of defects or defect probability) but also the inspection effort (often measured by the number of source code to be inspected).

To evaluate such effort-aware approaches, Mende et al. [46] propose a new metric Δopt . Δopt is defined as the area between the predicted model and optimal model, which results in a scalar value as shown in Fig 1. In Fig 1, the horizontal axis is the cumulative percentage of code churn of the changes (i.e., the inspection effort). The vertical axis is the cumulative percentage of defect-inducing changes found in selected changes. Δopt depicts deviations between predicted model and optimal model. Based on Δopt , they define $Popt$ as follows:

$$Popt = 1 - \Delta opt \quad (1)$$

The larger $Popt$ is (i.e., the smaller Δopt is), the closer the prediction model is to the optimal model. That is, a larger $Popt$ means a better performance of the effort-aware JIT defect prediction approach. $Popt$ is dependent on the number of defects in the data set. Consequently, normalising the $Popt$ for the prediction model is necessary for comparison among different data sets. The normalized $Popt$ computational formula can be defined as follows:

$$Popt(pre_model) = 1 - \frac{Area(optimal) - Area(pre_model)}{Area(optimal) - Area(worst)} \quad (2)$$

$Area(optimal)$, $Area(pre_model)$, and $Area(worst)$ represent the area under the curve corresponding to the optimal model, the prediction model and the worst model, respectively.

Recall is another performance indicator widely used to measure the performance of the effort-aware defect prediction. It measures how many percentage of defects could be identified when a certain percentage (e.g., 20%) of the effort is allocated. It is calculated as follows:

$$Recall = \frac{m}{n} \quad (3)$$

where m represents the number of identified defects, and n represents the total number of defects in test data set. *Recall* depends on the allocated inspection effort, and the more effort is allocated, the greater recall will be. The recall value when allocated effort is just enough to inspect 20% of the source code is widely used because research results suggest that 20% of the files (costing 20% of inspection effort) contain around 80% of the defects [47].

Deep learning in software engineering

There are many deep learning algorithms, such as Deep Belief NetWork (DBN)[48], Recurrent Neural Networks (RNN)[49], Convolutional Neural Networks (CNN)[50] and Long-Short Term Memory (LSTM) [51]. Deep learning has been applied in many fields of software engineering. Raychev et al. [32, 33] use the recurrent neural network (RNN) model to predict the probability of $(i + 1)$ -st word for code completion. White et al. [52] leverage deep software language model (RNN) in code suggestion. Mou et al. [53] proposed a tree-based convolutional neural network (TBCNN) for programming language processing, which can detect the structural features of program. Ling et al. [54] propose an attention model for code generation. Lam et al. [55] propose a new approach that leveraged deep neural network (DNN) in combination with rVSM to improve the accuracy of bug localization.

Deep learning has also been applied in defect prediction [11, 42, 56]. Yang et al. employ Deep Belief NetWork to features detection for defect prediction [42]. They use DBN for advanced features integration from fourteen change metrics. Then, they construct a logistic regression classifier to predict the changes as buggy or clean. Our approach is different from

Fig 1. Schematic diagram of an effort-based cumulative lift chart.

<https://doi.org/10.1371/journal.pone.0211359.g001>

their approaches in that they take the defect prediction as a binary classification problem, whereas we take it as a regression problem. Besides that, they connect the Deep Belief Network to a logistic regression classifier whereas our approach is a pure network.

Wang et al. [11] leverage the deep learning technology to learn semantic features directly from source code for defect prediction. They utilize Deep Belief Network to generate semantic features from token vectors extracted from programs' java Abstract Syntax Tree (AST). After that, they build a defect prediction model and perform defect prediction. They use the semantic features instead of the traditional features (Halstead features [57] and Mood features [58]) for the defect prediction. Our defect prediction use the change metrics that is often selected by experts for defect prediction as well.

Li et al. use Convolutional Neural Networks (CNN) for effective feature generation from source code with semantic and structural information to build a defect prediction model [4]. They adopt word embedding and combine the CNN-learned features with traditional features. However, they regard the defect prediction as a classification problem as well and inspection effort is not considered.

As a conclusion, our approach is different from the existing neural network based defect prediction approaches in the following aspects. First, Wang et al. [11] and Li et al.'s [4] defect prediction approaches are not effort-aware (i.e., their approaches don't take the effort of code inspection into account) whereas our approach is effort aware. Wang et al. [11] and Li et al.'s [4] approaches take the defect prediction as a binary classification. Our defect prediction approach regard the defect prediction as the ranking problem. The output of our NNR approach is a regression value, i.e., the defect probability of a change. Then we rank the changes according to the defect probability of a change divided by the code churn (i.e., $la + ld$). Second, Wang et al. [11] and Li et al.'s [4] approaches perform defect prediction on the file-level whereas our defect prediction approach is on the fine granularity (i.e., change-level).

Approach

Overview

An overview of the proposed approach is presented in Fig 2. The approach is composed of three steps: train a *NNR model*, make defect prediction for new changes based on the resulting *NNR model* and rank code changes for manual code inspection. To train the *NNR model*, we preprocess training data by applying logarithm, normalization and sampling. At prediction phase, we preprocess the metrics of the given code change and feed them into the trained neural network. The network generates the probability for the code change that contains defects. Based on the probability as well as the inspection effort for the code changes, we rank such changes so that inspecting them in order could find more buggy code changes with limited inspection effort. In the following sections, we present each of the key steps of the proposed approach.

Metrics for code changes

The public data set for JIT defect prediction is available at <http://research.cs.queensu.ca/~kamei/jittse/jit.zip>, shared by Kamei [13]. They generate the data set as follows. First, they extract information from the version control system (e.g., CVS repositories) of the projects and combine it with bug reports. Second, they use the heuristics to group related one-files commits into one software change. Third, they use the SZZ algorithm [59] and Approximate SZZ(ASZZ) algorithm to identify and recover the defect-inducing changes. In addition, they process the data as follows. (1) They deal with collinearity by manually removing the most highly correlated factors and using the stepwise variable selection. (2) They deal with skew by applying a standard log transformation. (3) They perform resampling of the training data to deal with data imbalance. They randomly delete the majority class (i.e., non-defect-inducing changes) to make the number of the majority class equal to the number of the minority class (i.e., defect-inducing changes).

The public data set contains 14 metrics of code changes. Such metrics are briefly introduced in Table 1 and they are concerned with five dimensions [13]. The first dimension is diffusion, concerning the number of modified subsystems (NS), the number of modified directories (ND), the number of modified files (NF) and the distribution of modified code across each file (Entropy). The second dimension is size, concerning the lines of code added (LA), the lines of code deleted (LD) and the lines of code in a file before the change (LT). The third dimension is purpose, only concerning FIX. Yin et al.'s [60] research shows that the change to fix a defect is more likely to introduce a new defect and incorrect bug-fixes will also introduce a new defect. FIX is whether or not the change is a defect fix. The fourth dimension is history, concerning the number of developers that changed the modified files (NDEV), the average time interval between the last and the current change (AGE) and the number of unique changes to the

Fig 2. Overview of our proposed approach.

<https://doi.org/10.1371/journal.pone.0211359.g002>

modified files (NUC). The fifth dimension is experience, concerning developer experience (EXP), recent developer experience (REXP), and subsystem experience (SEXP). Mockus and Weiss [19] argue that a programmer with higher experience will reduce their likelihood of inducing defect in a change. Developer experience (EXP) is measured by the number of changes made by the developer before current change. Recent developer experience (REXP) is measured by the total experience of the developers in term of change, weighted by their age. Subsystem experience (SEXP) measures the number of changes the developer made in the past to the subsystems that are modified by the current change.

Table 1. Metrics of code changes.

Dimension	Metric Name	Description
Diffusion	NS	The number of modified subsystems
	ND	The number of modified directories
	NF	The number of modified files
	Entropy	Distribution of modified code across each file
Size	LA	Lines of code added
	LD	Lines of code deleted
	LT	Lines of code in a file before the change
Purpose	FIX	Whether or not the change is a defect fix
History	NDEV	The number of developers that changed the modified files
	AGE	The average time interval between the last and the current change
	NUC	The number of unique changes to the modified files
Experience	EXP	The developer experience in terms of number of changes
	REXP	Recent developer experience
	SEXP	Developer experience on a subsystem

<https://doi.org/10.1371/journal.pone.0211359.t001>

Data preprocessing

We preprocess training data by applying logarithm, normalization and sampling.

Natural logarithm. Log transformation is widely used to make highly skewed distributions data sets less skewed [61]. It can make data conform to normality, reduce variability of data, and make patterns more visible [61]. We take the natural logarithm of each change metrics with the following formula:

$$x' = \ln x \quad (4)$$

Note that the transformation is not applied to FIX because it is boolean, i.e., it is either zero (no) or one (yes).

Normalization. Data normalization is widely used to transform data values to limit their changing range to a small range, e.g., from zero to one. Normalization is particularly useful for neural networks algorithms because it can reduce the estimation errors and decrease calculation time needed in training process [62]. Normalization can also reduce the chances of getting stuck in locally optima solution. One of the widely used normalization is Min-max normalization [63] that performs a linear transformation on the original data. Min-max normalization is defined as follows:

$$Norm(x) = \frac{x_i - \min(x)}{\max(x) - \min(x)} \quad (5)$$

where $\min(x)$ and $\max(x)$ represent minimum value and maximum value of a change metric x , respectively. x_i represents a raw value of this metric. Min-max normalization maps a value x_i to $norm(x)$ in the range $[0, 1]$. Such normalization can preserve the relationship among the raw data values [64]. It does not change the distribution of the data set.

In our data set, the range of change metrics is not uniform. NS, NM, NF, NDEV, PD, EXP, REXP and SEXP are raw data with different ranges [13] whereas other metrics have been normalized to $[0, 1]$. To unify the data format, we normalize such raw data with Min-max Normalization. We choose the Min-max normalization data set because of its high accuracy, low complexity and high learning speed [65].

Sampling. The training data is imbalanced because most of the changes are defect free whereas only a small percent of changes contain defects. The imbalance might reduce the accuracy of the prediction. To this end, we should ensure that the number of buggy changes is equal to the number of buggy free changes in the training data. We employ under-sampling technique [66] to ensure the equality of the numbers. Under-sampling method uses a subset of the majority class by only removing borderline and noisy majority examples to train the classifier. Since many majority class are ignored, the training set becomes more balanced [66].

Modelling

The key of the proposed approach is to predict how likely a given code commit contains defects. The prediction could be represented as a mapping:

$$y = f(cm) \quad (6)$$

$$y \in [0, 1] \quad (7)$$

where cm is the given code commit and y is the possibility for cm to contain defects.

As introduced in Metrics for Code Changes Section, code commits are often represented as a set of metrics in defect prediction.

$$x(cm) = \langle NS(cm), NF(cm), Entropy(cm), LT(cm), FIX(cm), \quad (8)$$

$$NDEV(cm), AGE(cm), NUC(cm), EXP(cm), SEXP(cm) \rangle \quad (9)$$

where $NS(cm)$, $NF(cm)$, $Entropy(cm)$, $LT(cm)$, $FIX(cm)$, $NDEV(cm)$, $AGE(cm)$, $NUC(cm)$, $EXP(cm)$, $SEXP(cm)$ are code metrics introduced in Metrics for Code Changes Section. As a result, the model of defect prediction can be represented as:

$$y = g(x(cm)) \quad (10)$$

Up to date, the mapping g is not yet full understood. Existing approaches [13] employs statistics based machine learning techniques (e.g., linear regression) to search for such mappings that best fit a given corpus of labelled data. In this paper, we try to learn the mapping with deep learning techniques as introduced in the following section.

Neural networks based defect prediction

We propose a neural network and deep learning based approach for effort-aware JIT defect prediction. The neural network is presented in Fig 3. It's a fully connected neural network. It consists of one input layer, one hidden layer and one output layer. $X = \langle NS, NF, \dots, SEXP \rangle$ is the input of the neural network, y is the predicted value of fully connected neural network (i.e., defect probability), and $w_{ns}, w_{sm}, \dots, w_{mf}$ are the weights for different input of the neurons. In total we have a connected neural network with three *dense* layers. We use ten change metrics (i.e., all fourteen change metrics except ND, REXP, LA and LD) as the input of the neural

Fig 3. Overview of the neural network.

<https://doi.org/10.1371/journal.pone.0211359.g003>

network. Its output is the defect probability for changes. Consequently, the dimension of the input of the neural network is 10 and the dimension of output is 1. For the first layer, we set `input_dim = 10`, `output_dim = 20`, `kernel_initializer = 'uniform'` and activation function = 'tanh'. For the second layer, we set `output_dim = 10`, `kernel_initializer = 'normal'` and activation function = 'relu'. For the last layer, `output_dim = 1` and `kernel_initializer = 'normal'`. We preprocess ten change metrics (i.e., all fourteen change metrics except ND, REXP, LA and LD) and feed them to the input layer of the neural network. NF and ND, REXP and EXP are highly correlated, high correlated features may decrease the performance of defect prediction. So we exclude the ND and REXP and use the change metrics NF and EXP instead. LA and LD are highly correlated. Nagappan and Ball [67] argue that relative churn metrics perform better than absolute metrics when predicting defect density. So, we remove the change metrics LA and LD and they will be used for ranking. The output of the neural network is the defect probability of the input changes. We employ Back Propagation (BP) algorithm to tune the weights with training data.

We use the training data to train the model. After that, we get a resulting model in each iteration of the k-fold cross validation. The `mean_squared_error` of the resulting models varies from 0.043 to 0.014. The `mean_absolute_error` ranges from 0.067 to 0.045 suggesting that the resulting models are quite accurate. To facilitate replication, we publish the source code of our model and the trained model at <https://github.com/donaldjoe/Effort-Aware-and-Just-in-Time-Defect-Prediction-with-Neural-Network>.

Effort-aware ranking

We compute the ranking benefit cost ratio $BCR(c)$ for a given change as follows:

$$BCR(c) = \frac{P(c)}{Effort(c)} \quad (11)$$

where $P(c)$ is defect probability predicted by the neural network, $Effort(c)$ is the required inspection effort measured by code churn (i.e., `la + ld`) [13]. For a set of changes, we rank them according to their $BCR(c)$ in descending order.

Evaluation

To validate the model, we follow Kamei et al. [13], Yang et al. [18] and Huang et al. [36] to conduct a 10-fold cross-validation technique as data analysis method and divide the data set into training data and testing data. In this section, we present the evaluation of the proposed approach on a publically available data set. The major purpose of the evaluation is to investigate whether the proposed approach can outperform existing approaches. To this end, we compare the proposed approach against the state-of-the-art approaches, i.e., EALR model, LT model and CBS model. Kamei et al.'s EALR model, Yang et al.'s LT model, and Huang et al.'s CBS model are the well-known defect prediction approaches. Such approaches are the just in time (i.e., make defect prediction on the change level) and effort-aware (i.e., taking the effort of code inspection into account) defect prediction approaches that are compared to the proposed approach.

Subject applications

We evaluate the selected approaches on six open-source subject projects, i.e., Bugzilla, Columba, Eclipse JDT, Eclipse Platform, Mozilla and PostgreSQL. We perform our effort-aware JIT defect prediction approach on a well-known data set created by Kamei et al. [13] (<http://research.cs.queensu.ca/kamei/jittse/jit.zip>). The data set is composed of changes from

six open-source projects, i.e., Bugzilla (<http://msr.uwaterloo.ca/msr2007/challenge/index>), Columba (<rsync://columba.cvs.sourceforge.net/cvsroot/columba/>), JDT (<http://msr.uwaterloo.ca/msr2008/challenge/index.html>), Platform (<http://msr.uwaterloo.ca/msr2008/challenge/index.html>), Mozilla (<http://msr.uwaterloo.ca/msr2007/challenge/index>) and PostgreSQL (<rsync://anoncvs.postgresql.org/cvsroot/pgsql/>). The data for Bugzilla and Mozilla are provided by MSR 2007 Mining Challenge, the data for Eclipse JDT and Platform are provided by MSR 2008 Mining Challenge, and the data for Columba and PostgreSQL are generated by combining the changes from the CVS repositories and bug reports from the bug tracking system (i.e., bugzilla) [13]. The data set is created by Kamei [13] and used by Yang et al. [18], and Huang et al. [36].

Table 2 gives the statistics of the six projects. The first column is the subject project name. The second column is the period time of the changes. From the third column to the sixth column present the total changes, percentage of defect-inducing defects, average of LOC per change, and number of the files modified in a code change, respectively. The language of the projects involves Perl, Java, C++, and Rubby. The number of total changes ranges from 4455 to 98275. The percentage of defects ranges form is 5% to 36%.

Process

An overview of our experiment process of evaluation is shown in Fig 4. 10-fold cross-validation technique [68] has been used in previous studies [13, 18, 36]. In consistent with previous studies, in our paper, we adopt 10-fold cross-validation technique to evaluate the performance of our defect prediction approach (i.e., NNR) against the state-of-the-art approaches (i.e., EALR, LT, and CBS) on the data set of each subject project. The process is presented in Fig 4. First, we preprocess the data set (it has been discussed in Data Preprocessing Section, i.e., applying logarithm, normalization and sampling). Second, we randomize the data set and randomly divide the data of each subject project into ten equal subsets. At each fold of the 10-fold cross-validation, we use one subset of the data as testing data (*testingData*) and the remaining nine subsets as training data (*trainingData*). Each subset is used as testing data once. For each fold, the evaluation is conducted as follows:

1. First, we train the proposed approach (i.e., NNR) with *trainingData* and get the *trainedModel*.
2. Second, we evaluate the *trainedModel*, EALR, LT and CBS on *testingData* independently.
3. Third, we calculate recall and popt value for each of the evaluated approaches.

The whole process is repeated 10 times in total and has 10 prediction value to diminish the bias in the data set splits. The average recall and popt value for each approach is calculated.

Table 2. Statistics of the studied data sets.

Project	Period	Total Changes	% of Defects	Mean LOC per change	Modified Files per Change
Bugzilla	08/1998–12/2006	4620	36%	37.5	2.3
Columba	05/2001–12/2007	4455	31%	149.4	6.2
Eclipse JDT	05/2001–12/2007	35386	14%	71.4	4.3
Eclipse Platform	05/2001–12/2007	64250	14%	72.2	4.3
Mozilla	01/2000–12/2006	98275	5%	106.5	5.3
PostgresSQL	07/1996–05/2010	20431	25%	101.3	4.5

<https://doi.org/10.1371/journal.pone.0211359.t002>

Fig 4. Process of the evaluation.

<https://doi.org/10.1371/journal.pone.0211359.g004>

Recall is calculated according to Eq 3 (in Effort-Aware Defect prediction and Performance Indicators Section). Popt is calculated according to Eqs 1 and 2 (in Effort-Aware Defect prediction and Performance Indicators Section).

We run the experiments on a personal computer: Intel Core i7-6700 CPU 3.4GHz, 16GB RAM. We use keras (<http://keras.io>) to build our neural network model.

Results

We carry out 10-fold cross-validation on the data set and compare the proposed approach against the state-of-the-art approaches, i.e., EALR [13], LT [18], and CBS [36].

Evaluation results are presented in Tables 3 and 4. Each row of the tables represents a subject application except that the last rows represent the average performance on all subject applications. Columns of these tables represent different approaches, i.e., EALR, LT, CBS and NNR (the proposed approach). Table 3 presents the recall of the approaches on six open-source projects. The recall is calculated when the allocated effort is just enough to inspect 20% of the source codes. The highest recall on each project is shown in bold. Table 4 presents the popt value of the approaches on six open-source projects. The highest popt are highlighted in bold as well. The distribution of recall and popt are depicted in Figs 5 and 6 respectively.

From such tables and figures, we make the following observation:

- First, the proposed approach (i.e., NNR) outperforms the state-of-the-art approaches (i.e., EALR, LT, CBS) in both recall and *popt*. The proposed approach improves the average recall

Table 3. Recall of the approaches.

Project	EALR	LT	CBS	Our Approach
Bugzilla	37.0%	49.5%	56.2%	75.4%
Columba	39.9%	62.0%	52.6%	78.3%
Eclipse JDT	20.1%	57.4%	54.8%	72.7%
Eclipse Platform	25.9%	50.9%	61.0%	73.7%
Mozilla	14.1%	37.5%	43.3%	55.0%
PostgresSQL	24.3%	54.2%	49.4%	68.5%
Average	26.9%	51.9%	53.1%	69.6%

<https://doi.org/10.1371/journal.pone.0211359.t003>

Table 4. Popt of the approaches.

Project	EALR	LT	CBS	Our Approach
Bugzilla	68.5%	76.0%	75.1%	90.1%
Columba	60.2%	83.3%	64.2%	90.6%
Eclipse JDT	46.6%	79.4%	65.6%	85.8%
Eclipse Platform	53.0%	77.2%	70.5%	85.5%
Mozilla	45.3%	66.5%	61.8%	75.4%
PostgreSQL	48.6%	80.7%	62.0%	84.6%
Average	53.7%	77.2%	66.2%	85.3%

<https://doi.org/10.1371/journal.pone.0211359.t004>

by 15.6% compared to the state-of-the-art approaches. The average recall achieved by EALR, LT, CBS and NNR is 26.9%, 51.9%, 53.1% and 69.6% respectively. The proposed approach also improves the average *popt* by 8.1% compared to the state-of-the-art approaches. The average *popt* achieved by EALR, LT, CBS and NNR are 53.7%, 77.2%, 66.2% and 85.3% respectively.

- Second, the proposed approach outperforms the state-of-the-art approaches on each of the subject projects. Generally, the results suggest that the proposed approach outperforms the others regardless of the involved projects.

Threats to validity

A threat to the construct validity is the suitability of the performance evaluation metrics (recall and *popt*) of the defect prediction approach. They are employed because they have been widely

Fig 5. Distribution of recall (beanplot).

<https://doi.org/10.1371/journal.pone.0211359.g005>

Fig 6. Distribution of popt (beanplot).

<https://doi.org/10.1371/journal.pone.0211359.g006>

used in previous effort-aware JIT defect prediction and haven been proved to be effective [13, 18].

A threat to the internal validity is the way we set the cut-off value. The value indicates the allocated inspection effort. It's represented as the percentage of source code that could be inspected with the allocated inspection effort. Consequently, the greater the cut-off value is, the greater recall would be. In our paper, we use 0.2 as the cut-off value to calculate the recall. The reasons are explained as follows. First, it is widely used by existing approaches [13, 18]. Second, Ostrand et al. found that around 20% lines of code may contain 80% fault [47]. Consequently, inspecting 20% of the most error-pron source code can hopefully find 80% of the defects.

The first threat to external validity is that the data set we used may be incomplete. This may affect the accuracy of our experimental results. The data set we used in our study was generated by Kamei et al. [13]. Kamei et al. [13] establish links between every defect fix and the source code change by using SZZ algorithms [69] and approximate SZZ (ASZZ) algorithms to generate the data sets. The SZZ and ASZZ algorithms may miss some links. The defect-inducing changes found may be not complete [70]. There may be some errors in the processing of extraction of defect-inducing changes. Despite all these, we still use this data set because the data set is the only public available change-level data set and it is widely used in the effort-aware JIT defect prediction.

The second threat to external validity is the limited number of subject applications. We evaluate our proposed approach on six open-source projects. Our study results may not be generalized to the other software projects.

Conclusion

In this paper, we propose a deep learning based approach for effort-aware just-in-time defect prediction. Based on sample changes, the approach trains a neural network to predict how likely a given change may contain defects. Given a set of changes to be inspected, the proposed approach ranks such changes according to their risky of defects (predicted by the trained neural network) divided by the required inspection effort (measured by the number of lines modified by the change). The key idea of the proposed approach is that we can exploit neural network and deep learning to select useful features for defect prediction because they have been proved excellent at selecting useful features for classification and regression.

Evaluation results on a well-known data set suggest that the proposed approach outperforms the state-of-the-art approaches. It improves the average recall and popt by 15.6% and 8.1%, respectively. We also notice that it outperforms the state-of-the-art approaches on each of the subject projects.

Author Contributions

Conceptualization: Lei Qiao.

Data curation: Lei Qiao.

Formal analysis: Lei Qiao.

Methodology: Lei Qiao.

Project administration: Lei Qiao.

Resources: Lei Qiao.

Software: Lei Qiao.

Validation: Lei Qiao, Yan Wang.

Visualization: Lei Qiao, Yan Wang.

Writing – original draft: Lei Qiao.

Writing – review & editing: Lei Qiao, Yan Wang.

References

1. Yu T, Wen W, Han X, Hayes J. ConPredictor: Concurrency Defect Prediction in Real-World Applications. *IEEE Transactions on Software Engineering*. 2018; p. 1–1. <https://doi.org/10.1109/TSE.2018.2791521>
2. Zhiqiang L, XiaoYuan J, Xiaoke Z. Progress on approaches to software defect prediction; 2018. Available from: <http://digital-library.theiet.org/content/journals/10.1049/iet-sen.2017.0148>.
3. Niu N, Bhowmik T, Liu H, Niu Z. Traceability-enabled refactoring for managing just-in-time requirements. In: 2014 IEEE 22nd International Requirements Engineering Conference (RE); 2014. p. 133–142.
4. Li J, He P, Zhu J, Lyu MR. Software Defect Prediction via Convolutional Neural Network. In: 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS); 2017. p. 318–328.
5. 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(4):485–496. <https://doi.org/10.1109/TSE.2008.35>
6. Tantithamthavorn C, McIntosh S, Hassan AE, Matsumoto K. An Empirical Comparison of Model Validation Techniques for Defect Prediction Models. *IEEE Transactions on Software Engineering*. 2017; 43(1):1–18. <https://doi.org/10.1109/TSE.2016.2584050>
7. Nam J, Fu W, Kim S, Menzies T, Tan L. Heterogeneous Defect Prediction. *IEEE Transactions on Software Engineering*. 2017; p. 1–1.

8. Jiang T, Tan L, Kim S. Personalized defect prediction. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE); 2013. p. 279–289.
9. Jing XY, Ying S, Zhang ZW, Wu SS, Liu J. Dictionary Learning Based Software Defect Prediction. In: Proceedings of the 36th International Conference on Software Engineering. ICSE 2014. New York, NY, USA: ACM; 2014. p. 414–423. Available from: <http://doi.acm.org/10.1145/2568225.2568320>.
10. Wang H, Khoshgoftar TM, Napolitano A. Software measurement data reduction using ensemble techniques. *Neurocomputing*. 2012; 92:124–132. <https://doi.org/10.1016/j.neucom.2011.08.040>
11. Wang S, Liu T, Tan L. Automatically Learning Semantic Features for Defect Prediction. In: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE); 2016. p. 297–308.
12. Tan M, Tan L, Dara S, Mayeux C. Online Defect Prediction for Imbalanced Data. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. vol. 2; 2015. p. 99–108.
13. Kamei Y, Shihab E, Adams B, Hassan AE, Mockus A, Sinha A, et al. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*. 2013; 39(6):757–773. <https://doi.org/10.1109/TSE.2012.70>
14. Schröter A, Zimmermann T, Zeller A. Predicting Component Failures at Design Time. In: Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering. ISESE'06. New York, NY, USA: ACM; 2006. p. 18–27. Available from: <http://doi.acm.org/10.1145/1159733.1159739>.
15. Meneely A, Williams L, Snipes W, Osborne J. Predicting Failures with Developer Networks and Social Network Analysis. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering. SIGSOFT'08/FSE-16. New York, NY, USA: ACM; 2008. p. 13–23. Available from: <http://doi.acm.org/10.1145/1453101.1453106>.
16. Giger E, D'Ambros M, Pinzger M, Gall HC. Method-level bug prediction. In: Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement; 2012. p. 171–180.
17. Hata H, Mizuno O, Kikuno T. Bug Prediction Based on Fine-grained Module Histories. In: Proceedings of the 34th International Conference on Software Engineering. ICSE'12. Piscataway, NJ, USA: IEEE Press; 2012. p. 200–210. Available from: <http://dl.acm.org/citation.cfm?id=2337223.2337247>.
18. Yang Y, Zhou Y, Liu J, Zhao Y, Lu H, Xu L, et al. Effort-aware Just-in-time Defect Prediction: Simple Unsupervised Models Could Be Better Than Supervised Models. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. FSE 2016. New York, NY, USA: ACM; 2016. p. 157–168. Available from: <http://doi.acm.org/10.1145/2950290.2950353>.
19. Mockus A, Weiss DM. Predicting risk of software changes. *Bell Labs Technical Journal*. 2000; 5(2):169–180. <https://doi.org/10.1002/bltj.2229>
20. McIntosh S, Kamei Y. Are Fix-Inducing Changes a Moving Target? A Longitudinal Case Study of Just-In-Time Defect Prediction. *IEEE Transactions on Software Engineering*. 2017; p. 1–1. <https://doi.org/10.1109/TSE.2017.2693980>
21. Fukushima T, Kamei Y, McIntosh S, Yamashita K, Ubayashi N. An Empirical Study of Just-in-time Defect Prediction Using Cross-project Models. In: Proceedings of the 11th Working Conference on Mining Software Repositories. MSR 2014. New York, NY, USA: ACM; 2014. p. 172–181. Available from: <http://doi.acm.org/10.1145/2597073.2597075>.
22. Tantithamthavorn C, McIntosh S, Hassan AE, Matsumoto K. The Impact of Automated Parameter Optimization on Defect Prediction Models. *IEEE Transactions on Software Engineering*. 2018; p. 1–1. <https://doi.org/10.1109/TSE.2018.2794977>
23. Ghotra B, McIntosh S, Hassan AE. Revisiting the Impact of Classification Techniques on the Performance of Defect Prediction Models. In: Proceedings of the 37th International Conference on Software Engineering—Volume 1. ICSE'15. Piscataway, NJ, USA: IEEE Press; 2015. p. 789–800. Available from: <http://dl.acm.org/citation.cfm?id=2818754.2818850>.
24. Bennin KE, Keung J, Phannachitta P, Monden A, Mensah S. MAHAKIL: Diversity based Oversampling Approach to Alleviate the Class Imbalance Issue in Software Defect Prediction. *IEEE Transactions on Software Engineering*. 2017; p. 1–1. <https://doi.org/10.1109/TSE.2017.2731766>
25. Wu F, Jing XY, Sun Y, Sun J, Huang L, Cui F, et al. Cross-Project and Within-Project Semisupervised Software Defect Prediction: A Unified Approach. *IEEE Transactions on Reliability*. 2018; p. 1–17. <https://doi.org/10.1109/TR.2018.2804922>
26. Bowes D, Hall T, Petric J. Software defect prediction: do different classifiers find the same defects? *Software Quality Journal*. 2017; . <http://doi.acm.org/10.1007/s11219-016-9353-3>
27. Bennin KE, Toda K, Kamei Y, Keung J, Monden A, Ubayashi N. Empirical Evaluation of Cross-Release Effort-Aware Defect Prediction Models. In: 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS); 2016. p. 214–221.

28. Ryu D, Jang JI, Baik J. A Hybrid Instance Selection Using Nearest-Neighbor for Cross-Project Defect Prediction. *Journal of Computer Science and Technology*. 2015; 30(5):969–980. <https://doi.org/10.1007/s11390-015-1575-5>
29. Arisholm E, Briand LC, Fuglerud M. Data Mining Techniques for Building Fault-proneness Models in Telecom Java Software. In: *The 18th IEEE International Symposium on Software Reliability (ISSRE'07)*; 2007. p. 215–224.
30. Mende T, Koschke R. Effort-Aware Defect Prediction Models. In: *2010 14th European Conference on Software Maintenance and Reengineering*; 2010. p. 107–116.
31. Lou J, Jiang Y, Shen Q, Shen Z, Wang Z, Wang R. Software reliability prediction via relevance vector regression. *Neurocomputing*. 2016; 186:66–73. <https://doi.org/10.1016/j.neucom.2015.12.077>.
32. Raychev V, Vechev M, Yahav E. Code Completion with Statistical Language Models. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI'14*. New York, NY, USA: ACM; 2014. p. 419–428. Available from: <http://doi.acm.org/10.1145/2594291.2594321>.
33. Raychev V, Vechev M, Yahav E. Code Completion with Statistical Language Models. *SIGPLAN Not*. 2014; 49(6):419–428. <https://doi.org/10.1145/2666356.2594321>
34. Jayanthi R, Florence L. Software defect prediction techniques using metrics based on neural network classifier. *Cluster Computing*. 2018;. <https://doi.org/10.1007/s10586-018-1730-1>
35. Manjula C, Florence L. Deep neural network based hybrid approach for software defect prediction using software metrics. *Cluster Computing*. 2018;. <https://doi.org/10.1007/s10586-018-1696-z>
36. Huang Q. Dataset and source code for ICSME2017 paper “Supervised vs Unsupervised Models: A Holistic Look at Effort-Aware Just-in-Time Defect Prediction”; 2017. Available from: <https://doi.org/10.5281/zenodo.836352>.
37. Akiyama F. An Example of Software System Debugging. In: *IFIP Congress (1)*; 1971. p. 353–359.
38. Zhong S, Khoshgoftaar TM, Seliya N. Unsupervised learning for expert-based software quality estimation. In: *Eighth IEEE International Symposium on High Assurance Systems Engineering*, 2004. Proceedings.; 2004. p. 149–155.
39. Zhang F, Zheng Q, Zou Y, Hassan AE. Cross-Project Defect Prediction Using a Connectivity-Based Unsupervised Classifier. In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*; 2016. p. 309–320.
40. Fu W, Menzies T. Revisiting Unsupervised Learning for Defect Prediction. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. ESEC/FSE 2017*. New York, NY, USA: ACM; 2017. p. 72–83. Available from: <http://doi.acm.org/10.1145/3106237.3106257>.
41. Mishra B, Shukla KK. Defect Prediction for Object Oriented Software using Support Vector based Fuzzy Classification Model. *International Journal of Computer Applications*. 2012; 60(15):8–16.
42. Yang X, Lo D, Xia X, Zhang Y, Sun J. Deep Learning for Just-in-Time Defect Prediction. In: *2015 IEEE International Conference on Software Quality, Reliability and Security*; 2015. p. 17–26.
43. Aizenberg I, Moraga C. Multilayer Feedforward Neural Network Based on Multi-valued Neurons (MLMVN) and a Backpropagation Learning Algorithm. *Soft Computing*. 2007; 11(2):169–183. <https://doi.org/10.1007/s00500-006-0075-5>
44. Li M, Zhang H, Wu R, Zhou ZH. Sample-based software defect prediction with active and semi-supervised learning. *Automated Software Engineering*. 2012; 19(2):201–230. <https://doi.org/10.1007/s10515-017-0220-7>
45. Mende T, Koschke R. Revisiting the Evaluation of Defect Prediction Models. In: *Proceedings of the 5th International Conference on Predictor Models in Software Engineering. PROMISE'09*. New York, NY, USA: ACM; 2009. p. 7:1–7:10. Available from: <http://doi.acm.org/10.1145/1540438.1540448>.
46. Mende T, Koschke R, Leszak M. Evaluating Defect Prediction Models for a Large Evolving Software System. In: *2009 13th European Conference on Software Maintenance and Reengineering*; 2009. p. 247–250.
47. Ostrand TJ, Weyuker EJ, Bell RM. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*. 2005; 31(4):340–355. <https://doi.org/10.1109/TSE.2005.49>
48. Hinton GE, Salakhutdinov RR. Reducing the dimensionality of data with neural networks. *Science*. 2006; 313(5786):504–507. <https://doi.org/10.1126/science.1127647> PMID: 16873662
49. Elman JL. Finding structure in time. *Cognitive Science*. 1990; 14(2):179–211. [https://doi.org/10.1016/0364-0213\(90\)90002-E](https://doi.org/10.1016/0364-0213(90)90002-E)
50. Krizhevsky A, Sutskever I, Hinton GE. ImageNet Classification with Deep Convolutional Neural Networks. *Commun ACM*. 2017; 60(6):84–90. <https://doi.org/10.1145/3065386>

51. Hochreiter S, Schmidhuber J. Long Short-Term Memory. *Neural Comput.* 1997; 9(8):1735–1780. <https://doi.org/10.1162/neco.1997.9.8.1735> PMID: 9377276
52. White M, Vendome C, Linares-Vasquez M, Poshyvanyk D. Toward Deep Learning Software Repositories. In: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories; 2015. p. 334–345.
53. Mou L, Li G, Zhang L, Wang T, Jin Z. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence. AAAI'16. AAAI Press; 2016. p. 1287–1293. Available from: <http://dl.acm.org/citation.cfm?id=3015812.3016002>.
54. Ling W, Blunsom P, Grefenstette E, Hermann KM, Kociský T, Wang F, et al. Latent Predictor Networks for Code Generation. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7–12, 2016, Berlin, Germany, Volume 1: Long Papers; 2016. Available from: <http://aclweb.org/anthology/P/P16/P16-1057.pdf>.
55. Lam AN, Nguyen AT, Nguyen HA, Nguyen TN. Combining Deep Learning with Information Retrieval to Localize Buggy Files for Bug Reports (N). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE); 2015. p. 476–481.
56. Singh PD, Chug A. Software defect prediction analysis using machine learning algorithms. In: 2017 7th International Conference on Cloud Computing, Data Science Engineering—Confluence; 2017. p. 775–781.
57. Halstead MH. Elements of Software Science (Operating and Programming Systems Series). New York, NY, USA: Elsevier Science Inc.; 1977.
58. Harrison R, Counsell SJ, Niithi RV. An evaluation of the MOOD set of object-oriented software metrics. *IEEE Transactions on Software Engineering.* 1998; 24(6):491–496. <https://doi.org/10.1109/32.689404>
59. Śliwowski J, Zimmermann T, Zeller A. When Do Changes Induce Fixes? In: Proceedings of the 2005 International Workshop on Mining Software Repositories. MSR'05. New York, NY, USA: ACM; 2005. p. 1–5. Available from: <http://doi.acm.org/10.1145/1082983.1083147>.
60. Yin Z, Yuan D, Zhou Y, Pasupathy S, Bairavasundaram L. How Do Fixes Become Bugs? In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. ESEC/FSE'11. New York, NY, USA: ACM; 2011. p. 26–36. Available from: <http://doi.acm.org/10.1145/2025113.2025121>.
61. FENG C, WANG H, LU N, CHEN T, HE H, LU Y, et al. Log-transformation and its implications for data analysis. *Shanghai Arch Psychiatry.* 2014; 26. <https://doi.org/10.3969/j.issn.1002-0829.2014.02.009>
62. Sola J, Sevilla J. Importance of input data normalization for the application of neural networks to complex industrial problems. *IEEE Transactions on Nuclear Science.* 1997; 44(3):1464–1468. <https://doi.org/10.1109/23.589532>
63. Patro SGK, Sahu KK. Normalization: A Preprocessing Stage. *CoRR.* 2015;abs/1503.06462.
64. Jain YK, Bhandare SK. Min max normalization based data perturbation method for privacy protection. *International Journal of Computer and Communication Technology.* 2011; 2.
65. Jiaqi P, Yan Z, Simon F. The impact of Data Normalization on Stock Market prediction: Using SVM and Technical Indicators. In: 2016 2nd International Conference Soft Computing in Data Science (SCDS 2016); 2016. p. 72–78.
66. Liu XY, Wu J, Zhou ZH. Exploratory Undersampling for Class-Imbalance Learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics).* 2009; 39(2):539–550. <https://doi.org/10.1109/TSMCB.2008.2007853>
67. Nagappan N, Ball T. Use of relative code churn measures to predict system defect density. In: Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.; 2005. p. 284–292.
68. Wong TT, Yang NY. Dependency Analysis of Accuracy Estimates in k-fold Cross Validation. *IEEE Transactions on Knowledge and Data Engineering.* 2017; PP(99):1–1. <https://doi.org/10.1109/TKDE.2017.2740926>
69. Kim S, Zimmermann T, Pan K, Whitehead EJJ. Automatic Identification of Bug-Introducing Changes. In: 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06); 2006. p. 81–90.
70. Wang L, Sun X, Wang J, Duan Y, Li B. Construct Bug Knowledge Graph for Bug Resolution. In: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C); 2017. p. 189–191.