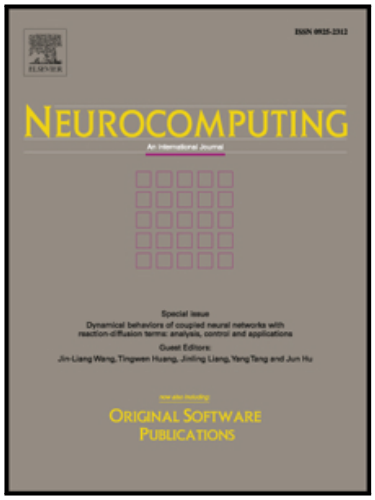
Journal Pre-proof



Deep Learning Based Software Defect Prediction

Lei Qiao, Xuesong Li, Qasim Umer, Ping Guo

PII: S0925-2312(19)31669-8
DOI: https://doi.org/10.1016/j.neucom.2019.11.067
Reference: NEUCOM 21595

To appear in: *Neurocomputing*

Received date: 20 June 2019
Revised date: 18 October 2019
Accepted date: 13 November 2019

Please cite this article as: Lei Qiao, Xuesong Li, Qasim Umer, Ping Guo, Deep Learning Based Software Defect Prediction, *Neurocomputing* (2019), doi: https://doi.org/10.1016/j.neucom.2019.11.067

Deep Learning Based Software Defect Prediction

Lei Qiao[a], Xuesong Li[a,*], Qasim Umer[a], Ping Guo[b,*]

[a]*School of Computer Science and Technology, Beijing Institute of Technology, Beijing, 100081, China*
[b]*Image Proc. and Patt. Recog. Lab., School of Systems Science, Beijing Normal University, Beijing, 100875, China*

Abstract

Software systems have become larger and more complex than ever. Such characteristics make it be very challengeable to prevent software defects. Therefore, automatically predicting the number of defects in software modules is necessary and may help developers efficiently to allocate limited resources. Various approaches have been proposed to identify and fix such defects at minimal cost. However, the performances of these approaches require significant improvement. Therefore, in this paper, we propose a novel approach that leverages deep learning techniques to predict the number of defects in software systems. First, we preprocess a publicly available dataset, including log transformation and data normalization. Second, we perform data modeling to prepare the data input for the deep learning model. Third, we pass the modeled data to a specially designed deep neural network-based model to predict the number of defects. We also evaluate the proposed approach on two well-known datasets. The evaluation results illustrate that the proposed approach is accurate and can improve upon the state-of-the-art approaches. On average, the proposed method significantly reduces the mean square error by more than 14% and increases the squared correlation coefficient by more than 8%.

Keywords: Software defect prediction, Deep learning, Software quality, Software metrics, Robustness evaluation.

*corresponding authors

Email addresses: qiaolei@bit.edu.cn (Lei Qiao), lixuesong@bit.edu.cn (Xuesong Li), qasimumer667@hotmail.com (Qasim Umer), pguo@bnu.edu.cn (Ping Guo)

1. Introduction

The complexity of modern software systems is increasing, and the resulting software applications often contain defects that can have severe negative impacts on the reliability and robustness of these applications [1]. A software defect is commonly defined as a deviation from the software specifications or requirements [2]. Such defects might lead to failures or produce unexpected results [2] [3]. To reduce failures and improve software quality, many software quality assurance activities (e.g., defect prediction, code review and unit testing) are employed. Such activities typically cost approximately 80% of the total budget of a project [4]. To minimize the cost, software engineers want to know which software modules contain more defects and inspect such modules first. As a result, software defect prediction techniques [5] have been proposed.

Software defect prediction techniques help identify software system modules that are more likely to contain defects [6]. Defect prediction techniques can be used to build models that rank software modules by the predicted number of defects, defect probability, or classification results [7]. This ranked list can reflect the priority for code inspection or unit testing and can thus be used to determine the order in which code should be inspected. Consequently, the developers can allocate the limited test resources to the code areas most likely to contain bugs. The resulting savings in labor and time costs [8] can reduce the overall cost of maintenance activities and maximize company profits[9].

Software defect prediction is a process that uses a model to predict the code areas that potentially contain defects [10]. The process of software defect prediction [11] includes three main steps: collecting a historical defect dataset; using the historical data to train a regression or classification model using machine learning or deep learning techniques; and applying the trained model to predict the number or probability of software defects. From the perspective of different dataset granularities, software defect prediction techniques can be divided into four categories: package level [12], file (modules) level [7], method level [13], and change level [9]. From the perspective of different metrics, it can be divided into two defect prediction categories: static and dynamic. Static defect prediction techniques mainly involve predicting the number of defects or the defect distribution using static software metrics [14]. Dynamic defect prediction techniques mainly involve predicting the distribution of system defects over time using the defect generation

time [10]. Most defect prediction techniques build defect prediction models that use traditional hand-crafted features, including Halstead's software volume metrics [15] and McCabe's cyclomatic complexity metrics [16]. Such approaches are based on statistical and machine learning theories. These approaches, which include support vector regression (SVR)[17], fuzzy support vector regression (FSVR) [18] and random forest (RF) [19] models, first build a regression or classification model and then use the model to predict the number or probability of defects for a given unit of source code. These approaches have significantly facilitated the prediction of software defects; however, their performances (e.g., mean square error and squared correlation coefficient) requires significant improvements [20].

While many defect prediction approaches have been proposed [10] [18] [21] [22] [23] [24] [25] [26] [27] [28], most of the research on defect prediction involves classification models [21]. These classification models classify a software module only into fault-prone or non-fault-prone. However, this type of defect prediction does not provide a specific number of defects. Predicting the defect probability of a given software module is not sufficient to help in practical software testing situations [29]. Moreover, allocating limited resources based solely on faulty or non-faulty judgements may result in an inefficient use of resources [26]. Some fault-prone modules may have more defects than other fault-prone modules; thus, inspecting and testing these modules requires additional effort [26].

In contrast, predicting the number of defects provides a specific estimate of the defects in a given module [26] [30]. Thus, the practitioners of software quality assurance activities can pay more attention to modules that have more defects and allocate their limited test resources more efficiently and optimally. Consequently, the developers can concentrate their testing efforts on modules that have more defects and accelerate the release schedule. Therefore, predicting the number of defects in a module is a good way to improve on predictions of whether a module is likely to be faulty or non-faulty. Although the existing approaches to defect prediction classify software modules as buggy or non-buggy modules, few predict the number of defects within modules [30] [31]. Such approaches exploit regression models for the prediction of the number of defects but require significant improvement in the process of software development. To this end, in this paper, our deep learning-based defect prediction model uses regression to predict the number of defects.

Deep learning has been applied in a wide variety of fields such as speech

recognition [32], natural language processing [33] and image processing [34], and has proven to be a powerful technique. To explore the power of deep learning for defect prediction and further improve the accuracy of defect prediction, in this paper, we propose a novel defect prediction approach, deep learning neural network-based defect prediction (DPNN). We employ deep learning for defect prediction because deep learning models can learn and capture the discriminative features from data automatically, thus resulting in a more accurate defect prediction model.

Wan et al.[35] conducted a large scale mixed qualitative and quantitative study and found that the top 3 preferred granularity levels of the practitioners are the feature (i.e., requirement or conceptual concerns proposed by customers/users), commit and component levels. Although defect prediction has finer granularity, it can be helpful in facilitating defect inspection and unit tests. A coarse granularity can help participants gain a good grasp of the overall quality of the software. In this paper, we make defect predictions at the module level.

The paper makes the following contributions:

- We propose a deep learning-based software defect prediction method.
- We evaluate the proposed approach with metrics on real-world software datasets and find that the proposed approach outperforms the state-of-the-art approaches.

The remainder of this paper is structured as follows. Section II reviews related works regarding software defect prediction. Section III describes the proposed approach. Section IV presents an evaluation of the proposed approach and compares it to previous approaches. Section V introduces the threats to the validity of defect prediction. Finally, Section VI provides conclusions and suggests potential future work.

2. Related Works

In this section, we introduce related works on software metrics, classification and regression models for software defect prediction, and deep learning and its applications.

2.1. Software Metrics

Software metrics[9] [15] [16] [36] [37] have been widely exploited in software defect prediction [38]. Zimmermann et al.[39] argued that combinations of complex metrics can be used to predict defects. They generated the Eclipse datasets for use with defect prediction models and made the datasets publicly available. Catal et al. [40] conducted a systematic review of previous software defect prediction research on various types of software metrics, methods, and datasets. They found that method-level metrics dominate in previous defect prediction research. Defect prediction that uses class-level metrics to determine software defects beyond acceptable levels should be applied more frequently because this approach can predict defects in the design phase [40]. D'Ambros [41] used different metrics (i.e., source code churn, the entropy of source code metrics, and process metrics) and built different classification models for defect prediction. Nam et al. [42] proposed two novel approaches, CLA and CLAMI, to label unlabeled datasets automatically using the magnitudes of metric values to facilitate defect prediction for unlabeled datasets. Ozakinci et al. [43] conducted a trend overview of software defect prediction early in the development life-cycle using process-based software metrics. Nam et al. [5] proposed heterogeneous defect prediction based on matching metrics to address limitations of cross-project defect prediction that cannot be addressed via heterogeneous metric sets.

A number of software defect prediction techniques that use software metrics have been proposed [7] [9] [21] [22] [23] [25] [27] [28] [44] [45]. Such approaches can be divided into two types: supervised defect prediction approaches and unsupervised defect prediction approaches. Supervised defect prediction approaches use historical datasets to train a defect prediction model [7]. Guo and Lyu [46] applied the pseudoinverse learning algorithm to build a software reliability growth model using the stacked generalization technique. They also adopted a support vector machine (SVM) to predict software quality [47]. Hata et al. [13] used an RF algorithm to build a method-level software defect prediction model based on historical method-level metrics. Unsupervised defect prediction approaches can predict defect proneness without requiring a defect dataset [45]. This approach can be used when a training dataset is insufficient or is not available [48]. Yang et al. [45] proposed an unsupervised approach that ranks the change metrics in descending order based on the reciprocal of the raw value of each change metric.

2.2. Classification Models for Software Defect Prediction

Software defect prediction models can help developers locate and fix more bugs promptly with less effort and can be classified as either classification or regression models. Most defect prediction models are classification models [21]. Lessmann et al. [21] proposed a framework to compare software defect classification predictions and conducted a large-scale empirical comparison of 22 classification models over 10 public datasets from the NASA Metrics Data repository. Ghotra et al. [25] replicated the research of Lessmann et al. [21], built different classification models using different classification techniques and tested them on three datasets. They found that defect prediction models vary significantly in performance and that significant differences exist among different classification techniques. Herbold et al. [49] replicated 24 approaches that used different classifiers (e.g., logistic regression, C4.5 Decision Tree, RF, etc.). They found that the logistic regression model proposed by Camargo Cruz et al. [50] was best at predicting fault-prone software. Jing et al. [10] proposed using a dictionary learning technique to predict software defects. They classified modules into buggy and non-buggy using the powerful classification capability of dictionary learning. Cross-project defect prediction requires some degree of homogeneity (i.e., different projects must be describable using the same metrics) between the projects used for training and those used for testing [5]. Zhang et al.[44] used a connectivity-based unsupervised classifier to solve this problem.

Laradji [51] studied the positive effects of combining feature selection and ensemble learning on the performance of defect classification. When building accurate classification models, it is necessary to select the features carefully. Tantithamthavorn et al. [28] studied the impacts of mislabeling datasets on classifier models. In another work, Tantithamthavorn et al. [52] [53] investigated classifier model performances and applied an automated parameter optimization technique called Caret. They found that Caret can improve the classifier performance significantly and that the resulting classifiers are a stable as those trained using the default settings. Parameter settings can have a large impact on the performance of defect prediction models. Ghotra et al. [54] studied the impact of feature selection techniques on the performance of defect classification models. They recommended applying feature selection techniques when building classification models for defect prediction.

He et al. [55] investigated the effects of using a simplified metric set to build defect predictors using both within-project and cross-project defect prediction. They found that naive Bayes models achieve good defect predic-

tion results. Tan et al. [11] conducted the first study that applied online change classification to improve defect prediction performance. To improve the performance of cross-project defect prediction, Zhang et al. [56] studied many composite algorithms that integrated multiple machine learning classifiers for defect prediction.

2.3. Regression Models for Software Defect Prediction

Whereas classification models have been widely studied in defect prediction, research on defect prediction using regression models is more limited [30] [31]. A regression model considers defect prediction as a ranking task. The software modules are ranked according to the number of predicted defects they contain. Predicting the number of defects explicitly in software modules is not easy. Therefore, an accurate regression model is necessary.

Mockus et al. [57] conducted the first study using a linear regression model to predict software failures on a change-level dataset. Kamei et al. [9] built an effort-aware defect prediction model by using linear regression on a dataset consisting of change metrics. Ohlsson et al. [58] proposed the Alberg diagram as a performance measure to assess the regression model performance. Yang et al. [59] proposed a learning to rank approach to optimize the performance measure of the ranking model. They use fault-percentile-average (FPA) and the defect percentage in the first 20% of the modules to evaluate the prediction models. Felix et al. [60] proposed a machine-learning-inspired (MACLI) approach using the predictor variables derived from defect acceleration to predict the number of defects in an upcoming product release and determine the correlation of each predictor variable with the number of defects. They found that only the average defect velocity shows a strong positive correlation with the true number of defects.

SVM was proposed by Vapnik et al. [61]. The theory of SVM is based on the idea of the structural risk minimization (SRM) approach [62]. The SVM was introduced to address classification problems. SVR is an extension of SVM used in solving linear and non-linear regression problems [63].

FSVR is commonly considered to be a combination of an SVR with fuzzy logic [64]. SVR considers that all the data points have the same importance in classification problems [65]. To reduce the sensitivity of the less important data, fuzzy logic (i.e., fuzzy membership) is introduced to each data point in SVR. Fuzzy memberships assign different membership values as weights to control the importance of the corresponding data point [65]. Yan et al. [18] employed FSVR to predict the number of defects in software and achieve

better performance. Fuzzifying the input to their regression approach can address problems with unbalanced datasets. Lee et al. [64] applied an FSVR model to predict the minimum departure from the nucleate boiling ratio (DNBR) in a reactor core to prevent fuel cladding from melting and causing a crisis. Zhang et al. [66] proposed a fuzzy density weight SVR (FDW-SVR) denoising algorithm that allocated a fuzzy priority to each sample according to its corresponding density weight.

Sun et al. [67] applied fuzzy SVM to the regression estimation problem. They construct a multi-layer SVM by combining fuzzy logic with the SVMs. In the first layer, they used a fuzzy membership function to the SVM; then, in the second layer, they used a generalized SVM. Liu et al. [68] proposed a three-domain fuzzy support vector regression model that integrates the kernel and fuzzy membership functions into a three-domain function that performs uncertain image denoising for a humanoid robot. Chen et al. [69] utilized a three-layer weighted fuzzy SVR model based on emotion-identification information to understand human intention. This model includes three layers: adjusted weighted kernel fuzzy c-means, fuzzy SVR, and weighted fuzzy regression. Their model also has the ability to reveal the weights assigned to each feature.

Quinlan proposed an approach to synthesizing decision trees and demonstrated through several practical applications that the technology for building decision trees is robust [70]. Seliya et al. [71] used C4.5 and RF decision tree classification algorithms in the context of cost-sensitive learning to build a software quality prediction model. Chen et al. [31] used decision tree regression (DTR) to build a defect prediction model to predict the number of defects in different scenarios (i.e., within- and cross-project defect scenarios). They found that the defect prediction models can obtain similar performance results in the within- and cross-project defect scenarios. Rathore et al. [30] applied the DTR to predict the number of defects in two scenarios: intra- and inter-release defect prediction. Yu et al. [29] used the DTR model as a baseline for defect prediction. Rathore et al. [26] conducted an empirical study on defect prediction models that predict the number of defects. They used the average absolute error, average relative error, and level-l measures as the performance criteria to evaluate the performances of the defect prediction models and found that the DTR achieved the best performance.

2.4. Deep Learning and Its Applications

Many deep learning algorithms have been proposed, including the convolutional neural networks (CNN)[72], recurrent neural network (RNN)[73], and long-short term memory (LSTM) [74]. Deep learning has been implemented in a variety of domains such as speech recognition [32], natural language processing [33] and image processing [34].

Currently, deep learning is becoming increasingly prevalent in the field of software engineering. Zhao et al. [75] proposed a new approach, DeepSim, to measure the functional similarities of code. They encoded code control-flow and data-flow into a matrix to create a semantic representation and then used a deep neural network (DNN) model to learn features from the matrix and conduct a binary classification. Ma et al. [76] proposed MODE, an automated neural network debugging technique. Similar to software debugging and regression testing, they conducted a model-state analysis to identify model bugs and performed training input selection. MODE can efficiently identify buggy neurons and fix model bugs. Guo et al. [77] used word embedding and an RNN to generate trace links. The word embedding learns word vectors; then, the RNN uses these word vectors to learn the sentence semantics. White et al. [78] used a deep learning software language model (i.e., a feed-forward network and an RNN) to make code suggestions. They also identified avenues for software engineering tasks using deep software language models to make predictions. They reported that deep learning techniques are applicable to source code files and found that deep learning can create high-quality models from a corpus of Java projects. Balog et al. [79] employed a neural network to solve programming-competition style problems from input-output examples. Tian et al. [80] applied a DNN to automatically generate test cases to perform automated testing of erroneous behaviors of DNN-controlled vehicles.

Huo et al. [81] applied a CNN to locate potential buggy source code based on a bug report. They used both lexical and program structural information to learn unified features from natural language and source code to perform bug localization. Their approach used a CNN to extract complete and semantic features. Gu et al.[82] propose a new DEEPAPI approach built with an RNN encoder-decoder model to generate an API suggestion sequence for a given API-related query. They were the first to apply an RNN to API sequence suggestions. White et al. [83] proposed a deep learning technique to detect code clones. Their approach automatically discovers discriminating features in source code. They suggested that all the content in the terms

and fragments of source code can be represented and used for clone detection. Xu et al. [84] proposed a deep learning approach to solve the problem of predicting semantically linkable knowledge units. They considered it as a multi-class classification problem rather than a traditional binary classification problem. They uses word embeddings and a CNN to capture the word-and-document-level semantics of the knowledge units.

Lam et al. [85] proposed a novel approach that combined a DNN and revised vector space model (rVSM) for bug localization in which they used the DNN to learn and relate the terms in bug reports and source files. Mou et al. [86] proposed a novel tree-based convolutional neural network (TBCNN) for processing programming languages in which the convolutional layer detects is used to detect structural features. Trees of different sizes and shapes can be dealt with by continuous binary trees and pooling. Gu et al. [87] proposed a novel deep neural network named the Code-Description Embedding Neural Network (CODEnn) to help developers perform code searches. The code snippets and corresponding nature language descriptions are represented as a high-dimensional vector space. They built a deep neural network to perform the code search. Hellendoorn et al. [88] proposed DEEPTYPE, a deep learning model, to provide type suggestions. This model learns which types occur in certain contexts and can then be used to predict variable and function type annotations. Henkel et al. [89] proposed an approach that transforms programs into a more suitable representation. They then used the trace abstractions obtained from a program as a representation for learning word embeddings.

Our approach differs from the existing approaches in that we are the first to exploit a fully connected neural network to conduct module-level based defect prediction by using software metrics.

3. Approach

In this section, we propose a deep learning-based model for software defect prediction. An overview of the proposed approach is presented next; then the details are presented in the remainder of this section.

3.1. Overview

An overview of the deep learning-based software defect prediction is depicted in Fig. 1, which shows that the proposed approach is composed of two steps: training a deep learning-based defect prediction model (i.e., *DPNN*

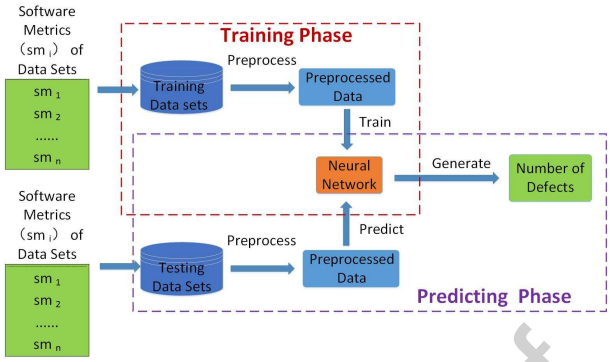


Fig. 1. Overview of the proposed approach.

model) and performing defect prediction for new modules based on the trained model. To predict the number of defects in a software module, we perform the following key steps.

- First, we collect and reuse datasets containing software metrics.
- Second, to train the *DPNN model*, we conduct some preprocessing operations (data log transformation and data normalization) on the given software metrics.
- Third, we train a specially designed deep learning neural network-based model to predict the number of defects in code.
- Finally, in the prediction phase, we perform preprocessing to acquire the software metrics and then input the modeled data to the trained regression model to predict the number of defects in each provided module.

Each of the key steps in the proposed approach is elaborated in the following sections.

3.2. Modeling

The key goal of the proposed approach is to predict the number of defects in software modules.

The model for defect prediction can be defined as a mapping:

$$y = f(m) \tag{1}$$

where m is a given module and y is the predicted number of defects in module m.

A module m consists of a set of software metrics and can be defined as follows:

$$m =< sm_1, sm_2, ..., sm_n > \tag{2}$$

where $sm_1, sm_2, ..., sm_n$ represent the software metrics of the module m.

The mapping f can be explored using different techniques. Existing defect prediction approaches apply statistical theories (e.g., linear regression or DTR) to search for the mappings that best fit a given dataset [9] [26]. In this paper, we attempt to find such mappings through deep learning techniques.

3.3. Data Preprocessing

To determine the number of defects in software modules, requires a preprocessing step to acquire the software metrics [9] [90] [91]. To accomplish this task, we apply natural logarithm transformation and perform data normalization on the extracted software metrics.

3.3.1. Natural logarithm

Log transformation converts the dataset distribution from highly skewed to less skewed [92]. We apply log transformation to each software metric due to the highly skewed distribution of our dataset. After log transformation, a module m can be defined as follows:

$$m =< sm_1{}', sm_2{}', ..., sm_n{}' > \tag{3}$$

$$SM =< ln(sm_1), ln(sm_2), ..., ln(sm_n) > \tag{4}$$

where $sm_1{}', sm_2{}', ..., sm_n{}'$ represents the log-transformed n software metrics, SM represents a set of log-transformed n software metrics, and $ln(sm_1), ln(sm_2), ..., ln(sm_n)$ represents the less skewed value of each software metric.

3.3.2. Normalization

Data normalization is a commonly used technique that transforms large data value ranges into small range values (or binary values). Notably, we perform min-max normalization [93] because it is a commonly used normalization approach due to its high accuracy and high learning speed [23] [94] [95] [96]. Moreover, min-max normalization does not change the dataset distribution[97].

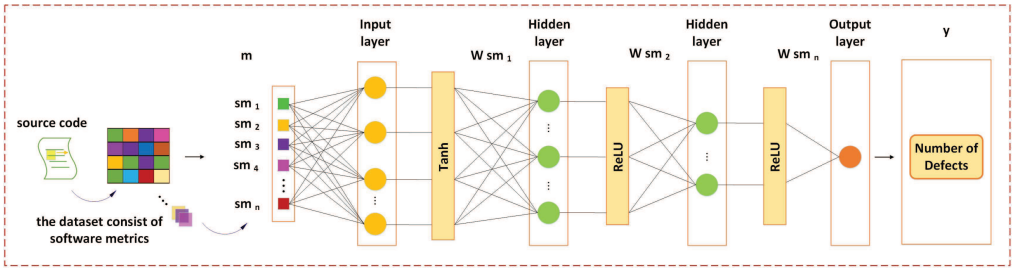


Fig. 2. Neural network overview.

We perform data normalization for the following reasons. First, the range of extracted software metrics varies greatly. Second, it is compulsory when using deep learning algorithms [98] [99]. Finally, it may reduce the estimation errors and calculation time (required in the training process) [100]. Min-max normalization can be defined as follows:

$$Normalization(sm_i^{'}) = \frac{sm_i^{'} - min(sm^{'})}{max(sm^{'}) - min(sm^{'})} \tag{5}$$

where $min(sm^{'})$ and $max(sm^{'})$ represent the minimum and maximum values of a software metric $sm^{'}$, respectively, $sm_i^{'}$ represents a logarithmic transformation value of the metric, and $Normalization(sm_i^{'})$ represents the normalized value of $sm_i^{'}$. After data normalization, module m can be defined as follows:

$$m =< sm_1^{''}, sm_2^{''}, ..., sm_n^{''} > \tag{6}$$

where m is a set of log-transformed and data normalized software metrics $(sm_i^{''})$.

3.4. Defect Prediction Using Deep Learning

We propose a deep learning neural network-based model for defect prediction. An overview of this fully connected neural network is shown in Fig. 2 consisting of one input layer, two hidden layers, and one output layer. We preprocess the software metrics and input them to the neural network (as presented in Eq. 6). The neural network predicts the number of defects y for each module, where W_{sm_1}, $W_{sm_2}...W_{sm_n}$ are the weights of different inputs to the neurons. We set the input dimensions, output dimensions, kernel initializer and activation function for each layer as follows:

- First layer: input_dim = 11, output_dim = 20, kernel_initializer = 'uniform' and activation function = 'Tanh'.
- Second layer, output_dim = 10, kernel_initializer = 'normal' and activation function = 'ReLU'.
- Third layer, output_dim = 6, kernel_initializer = 'normal' and activation function = 'ReLU'.
- Last layer, out_dim = 1 and kernel_initializer = 'normal'.

Notably, we set different input_dim values of the neural network for different datasets because the number of software metrics differ. Because the Medical Imaging System (MIS) dataset [1] and the NASA Metrics Data Program (MDP) for NASA PROMISE (KC2) dataset [2] have 11 and 21 software metrics, we specify 11 input_dim and 21 input_dim for them, respectively. The output of the neural network is the predicted number of defects in the software module.

We explored using different neural network model architectures. In our algorithm, the input and output dimensions are fixed. We can change the number of layers and the activation function of the neural network. For the number of layers, we try to use a strategy similar to a greedy search. We changed the number of layers from one to four and added one layer to the neural network architecture each time. Finally, we found that the above parameter setting achieves the best performance.

4. Evaluation

In this section, we evaluate the proposed approach against three state-of-the-art approaches, *SVR* [101], *FSVR* [18] and *DTR* [26], on two well-known datasets.

4.1. Research Questions

The evaluation investigates the following research questions:

- **RQ1**: Does the proposed approach outperform existing approaches?

[1] http://www.cse.cuhk.edu.hk/ lyu/book/reliability/
[2] http://promise.site.uottawa.ca/SERepository/datasets-page.html

- **RQ2**: How efficient is the proposed approach? How long does it take to train the neural network-based regression model, and how long does it take to perform defect prediction?

Research question RQ1 investigates the effectiveness of the proposed model against the *SVR*, *FSVR*, and *DTR* models. We choose the *SVR*, *FSVR*, and *DTR* models for comparison for the following reasons. First, these state-of-the-art approaches all provide automatic predictions of the number of defects in software modules. Second, to the best of our knowledge, these state-of-the-art approaches have been reported to be more accurate than other related approaches [18] [26] [101].

Research question RQ2 reflects the time complexity of the proposed approach. Generally speaking, training a deep learning model requires a long time. To compute the time complexity of the proposed approach, we recorded the training and testing (prediction) times to reveal the time efficiency of the compared approaches.

4.2. Datasets and Metrics

We reuse the software metrics extracted from the MIS dataset and the KC2 dataset. Notably, the selected software metrics (mentioned in Table 1 and 2) from both datasets (MIS and KC2) are the most commonly used software metrics for defect prediction [5].

We collected the MIS dataset from the compact disc that accompanies the "Handbook of Software Reliability Engineering" [102]. The MIS dataset is a commercial medical imaging system consisting of 390 records, 12 software metrics, and approximately 400,000 lines of code [103]. This application can be divided into 4,500 modules. It is written in Pascal, Fortran, assembly language, and PL/M (the Intel-8 programming language for microcomputers). The software metrics of MIS are briefly introduced in Table 1.

We collected the KC2 dataset from the PROMISE software dataset repository [104]. The KC2 dataset includes five dimensions, i.e., McCabe, Halstead, LineCount, Operator and Operand, and Branch. We used 21 software metrics of the KC2 dataset (e.g., LOC, V(G), EV(G), etc.) that are briefly introduced in Table 2.

Table 1 and 2 list the metrics in the MIS and KC2 dataset, respectively. The MIS dataset contains 12 metrics. These metrics include the total number of code lines (LOC), total code lines (CL),..., and Belady's bandwidth metric (BW). The KC2 dataset involves twenty-one software metrics and

Table 1. MIS Metrics

Software Metrics	***Description***
LOC	Total number of code lines
CL	Total code lines
TChar	Total number of characters
TComm	Total comments
MChar	Number of comment characters
DChar	Number of code characters
N	Halstead's program length
NE	Halstead's estimated program length
NF	Jensen's estimated program length
V(G)	McCabe's cyclomatic complexity
BW	Belady's bandwidth metric

five dimensions. The first dimension is McCabe. The second dimension is Halstead. The Halstead metric is commonly used for defect prediction [5]. The third dimension is line count, including Halstead's line count, Halstead's count of comment lines, Halstead's count of blank lines, Halstead's code and comment count for each module. The fourth dimension, operator operand, includes the number of unique operators, number of unique operands, total operators and total operands. The fifth dimension is branch, which considers only the total number of branches.

4.3. Performance Criteria

The proposed model is evaluated on the first 80% of the modules and the last 20% of the modules (mentioned in Section 4.4). Notably, we use the number of defects for evaluation and calculate the mean squared error (denoted as *MSE*) [105] and the coefficient of determination (denoted as R^2) [106] to evaluate the performance of the defect prediction model.

The number of defects is the number of defects contained in a software module. For a given module, our defect prediction model and other state-of-the-art approaches can predict the number of defects. The closer the number of predicted defects is to the actual number of defects contained in a module, the better the performance of the defect prediction model.

The *MSE* measures the average of the squared errors—the squared difference between the estimated parameter and the true value of the parameter [107]. The *MSE* measures the quality of a prediction model and better values are non-negative and closer to zero. Thus, a smaller *MSE* represents better prediction model performance [105]. Moreover, a smaller *MSE* means that the prediction model is closer to the optimal model [108] [109]. The *MSE* can be computed as follows [110]:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{7}$$

where y_i represents the vector of actual values of the variable, $\hat{y}_i$ represents the predicted variable values, and n is the number of data points [111].

R^2 measures the proportion of the variance in the dependent variable that is predictable from the independent variables [112], and it is calculated by squaring the correlation coefficient between the observed and predicted values in a regression model [112]. R^2 represents the strength of the relationship between an independent and dependent variable(s)[106], and it measures the performance of a model. Its values range from 0 to 1. An R^2 value between 0 and 1 shows the extent to which the dependent variable is predictable, which represents how well the regression model fits the datasets. An R^2 value closer to 1 indicates that the model has achieved a good fit to this problem and represents the strength of the relationship between an independent and dependent variable(s). A larger R^2 value represents a better model fit [113]. The R^2 can be computed as follows:

$$R^2 = \frac{\sum_{i=1}^{n}(\hat{y}_i - \bar{y})^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \tag{8}$$

where y_i represents the observed values of the dependent variable, $\bar{y}$ is the mean value, and $\hat{y}_i$ is the predicted value.

4.4. Process

Feng et al. [114] proposed that a small number of modules contain most of the defects. Similarly, Koru et al. [115] proposed that smaller modules are proportionally more defective than are larger ones. Some other empirical studies show that only a few (approximately 20 percent) of files contain nearly 80 percent of faults (i.e., Pareto principle, 80/20 rule) [114] [27] [116].

Table 2. Metrics of KC2

Metric Type	**Software Metrics**	***Description***
McCabe	LOC	McCabe's code line count
	V(G)	McCabe's "cyclomatic complexity"
	EV(G)	McCabe's "essential complexity"
	IV(G)	McCabe "design complexity"
Halstead	N	Halstead total operators + operands
	V	Halstead's "volume" for each module
	L	Halstead's program length
	D	Halstead's "difficulty"
	I	Halstead's "intelligence"
	E	Halstead's "effort"
	B	Halstead
	T	Halstead's time estimator
Line Count	LOCode	Halstead's line count
	LOComment	Halstead's count of comment lines
	LOBlank	Halstead's count of blank lines
	LOCodeAnd Comment	Halstead's code and comment count for each module
Operator Operand	Uniq_Op	Unique operators
	Uniq_Opnd	Unique operands
	Total_Op	Total operators
	Total_Opnd	Total operands
Branch	BranchCount	of the flow graph

Software testing engineers should allocate more resources to the modules that contain more defects.

We follow Yan et al. [18] and rank the dataset based on the number of defects in software modules in ascending order. Finally, we divide the datasets into the first 80% and last 20% of modules based on their number of defects. We use the first 80% of modules for training and the last 20% of modules for prediction. For testing data, we predict the number of defects of every module and then add the defects to calculate the total number of defects based on the rule: predicting the number of defects 4 and 5 is more accurate than predicting the number of defects 6 and 5 when two modules are input with 5 and 6 defects. Notably, we use cross-validation to evaluate the proposed approach on both datasets.

Cross-validation is used to flag problems such as overfitting or selection bias. To validate the defect prediction model, we conduct a k-fold (k = 10) cross-validation on two state-of-the-art publicly available datasets. In 10-fold cross-validation, the datasets are randomly divided into 10 equal subsets. For each fold of the 10-fold cross-validation, one subset is used as testing data, and the remaining nine subsets are used as training data. Thus, each subset is used as testing data only once. The following process is used for each fold of the evaluation:

- **Step 1**, we train the proposed approach with the training datasets and obtain the trained model.
- **Step 2**, we evaluate the trained models (the proposed approach, SVR, FSVR, and DTR) on the testing data.
- **Step 3**, we calculate the performance metrics (i.e., *MSE* and R^2) for each of the evaluated approaches.

4.5. Results

4.5.1. RQ1: Comparison with three State-of-the-Art Approaches

To answer research question RQ1, we conduct a 10-fold cross-validation on two well-known datasets and compare the proposed approach against the three state-of-the-art approaches (i.e., *SVR*, *FSVR*, and *DTR*). The evaluation results are presented in Tables 3, 4, 5, and 6.

Table 3 presents the performance results of *SVR*, *FSVR*, *DTR*, and the proposed approach on the complete dataset and we treat the whole dataset as the training dataset. The first column lists the approaches, and columns

Table 3. Experimental results on the complete dataset

Approach	***MIS-MSE***	***MIS-R^2***	***KC2-MSE***	***KC2-R^2***
SVR	47.35	0.32	0.13	0.193
FSVR	57.64	0.41	0.127	0.228
DTR	66.30	0.245	0.126	0.234
Proposed	**46.01**	**0.42**	**0.109**	**0.297**

Table 4. Total number of defects in the last 20% of the modules

Dataset	***Training Set***	***SVR***	***FSVR***	***DTR***	***Proposed Approach***
MIS-20%	2521	1924	2101	1629	**2472**
KC2-20%	205	172	179	58	**202**

2&3 and 4&5 present the *MSE* and R^2 performance results on the MIS and KC2 datasets, respectively. The rows show the performance results of *SVR*, *FSVR*, *DTR*, and the proposed approach. The *MSE* and R^2 of *SVR*, *FSVR*, *DTR*, and the proposed approach on the MIS dataset are (47.35, 0.32), (57.64, 0.41), (66.30, 0.245), and (46.01, 0.42), respectively. Similarly, the *MSE* and R^2 of *SVR*, *FSVR*, *DTR*, and the proposed approach on KC2 dataset are (0.13, 0.193), (0.127, 0.228), (0.126, 0.234), and (0.109, 0.297), respectively.

Table 4 presents the number of defect predictions achieved by the proposed approach compared to *SVR*, *FSVR*, and *DTR* on the last 20% of modules of the MIS and KC2 datasets. The first column lists the datasets. Column 2 lists the total number of defects in the last 20% of modules in the training datasets of the MIS and KC2, and columns 3–6 present the performance results of *SVR*, *FSVR*, *DTR*, and the proposed approach, respectively. The rows show the performance results of these approaches on the last 20% of modules of the MIS and KC2 datasets, respectively. We use the training data to train the defect prediction model. Then, the defect prediction model is used to predict the number of defects in the last 20% modules. We predict the number of defects of every module of the testing data. Finally, we add the defects to calculate the total number of defects.

Table 5. MSE of first 80% and last 20% subsets of MIS and KC2

Approach	*MIS-80%*	*MIS-20%*	*KC2-80%*	*KC2-20%*
SVR	5.9	128.79	0.017	4.292
FSVR	6.41	93.68	0.016	3.656
DTR	6.24	121.23	0.007	3.22
Proposed	**6.05**	**54.91**	**0.005**	**2.970**

Table 6. MSE for training and testing subsets

Approach	*MIS-Train*	*MIS-Test*	*KC2-Train*	*KC2-Test*
SVR	68.89	72.33	0.825	0.976
FSVR	60.85	63.61	0.885	0.989
DTR	57.8	63.61	0.922	1.36
Proposed	**46.56**	**51.07**	**0.754**	**0.958**

Table 5 presents the *MSE* results achieved by the proposed approach compared to *SVR*, *FSVR*, and *DTR* on the first 80% of modules and the last 20% of modules of the MIS and KC2 datasets. The first column presents the approaches, and columns 2&3 and 4&5 list the performance results of the first 80% modules and the last 20% modules on the MIS and KC2 datasets, respectively. The rows show the performance results of *SVR*, *FSVR*, *DTR*, and the proposed approach.

The MIS dataset contain 78 modules in the last 20% of modules. We randomly select 10 samples from these 78 modules as the testing dataset. Other modules and the first 80% of modules are regarded as the training dataset. This process is repeated 10 times for 10-fold cross-validation. We then calculate the average *MSE*. The same experiment is performed on the KC2 dataset. Table 6 presents the average *MSE* achieved by the proposed approach against *SVR*, *FSVR*, and *DTR* on the training and testing samples of the MIS and KC datasets. The first column lists the approaches, and columns 2&3 and 4&5 list the performance results with the training and testing samples on the MIS and KC2 datasets, respectively. The rows show the performance results of *SVR*, *FSVR*, *DTR*, and the proposed approach.

From Tables 3, 4, 5, and table 6, we can make the following observations:

- First, the proposed approach outperforms *SVR*, *FSVR*, and *DTR* on both the MIS and KC2 datasets. The *MSE* of MIS varies from 66.30 to 46.01, and the R^2 of MIS varies from 0.32 to 0.42. Similarly, the *MSE* of KC2 varies from 0.13 to 0.109, and the R^2 of KC2 varies from 0.193 to 0.297.

- Second, the proposed approach outperforms *SVR*, *FSVR*, and *DTR* by improving the number of defect predictions from 23 = (202 - 179) to 371 = (2472 - 2101).

- Third, the proposed approach outperforms *SVR*, *FSVR*, and *DTR* on MIS (last 20% of modules) and KC2 (last 20% of modules and first 80% of modules) datasets by 0.71 = (93.68 - 54.91) / 54.91, 0.4 = (0.007 - 0.005) / 0.005, and 0.08 = (3.22 - 2.97) / 2.97, respectively. Whereas, *SVR* performs marginally better than the proposed approach on the first 80% of modules of MIS by 0.03 = (6.05 - 5.9) / 5.9. However, the proposed approach performs better than the other approaches (*FSVR* and *DTR*) on the first 80% modules of MIS by 0.06 = (6.41 - 6.05) / 6.05 and 0.03 = (6.24 - 6.05) / 6.05, respectively.

- Finally, the proposed approach significantly outperforms the *SVR*, *FSVR*, and *DTR* models in terms of *MSE* on the training and testing sets of both the MIS and KC2 datasets. The performance improvement of the proposed approach against *SVR*, *FSVR*, and *DTR* on all training and testing sets is 0.24 = (57.80 - 46.56) / 46.56, 0.25 = (63.61 - 51.07) / 51.07, 0.09 = (0.825 - 0.754) / 0.754, and 0.02 = (0.976 - 0.958) / 0.958, respectively.

We note that our proposed approach leveraging the neural network for number of defect prediction is effective (i.e., our approach achieves lower *MSE* and higher R^2) compared to the state-of-the-art approaches and the improvement is achieved by our approach for the following reasons. First, in the fully connected neural network, a fully connected layer learns features from all the combinations of the features of the previous layer. The connection of each neuron to every neuron in the previous layer makes this process possible, and each connection has its own weight. Second, the neural network may use expressive, continuous-valued representations to learn and capture

the discriminative and useful features powerfully from a dataset consisting of software metrics. Third, we preprocess the dataset adequately. Fourth, we use cross-validation to reduce the bias of our defect prediction model.

Based on the preceding analysis, we can conclude that the proposed approach is accurate and significantly outperforms the state-of-the-art approaches.

4.5.2. RQ2: Efficiency of the Proposed Approach

To answer research question RQ2 (i.e., to investigate the efficiency of the deep learning-based regression model) we record the time cost of the training and prediction processes. The evaluation results suggest that the proposed approach is efficient. On average, training the proposed approach is completed within 3 minutes. Using the trained model, it takes 0.04 minutes to predict the number of defects. On average, the training process for *SVR*, *FSVR*, and *DTR* requires 0.08 minutes, 1.2 minutes and 0.9 minutes, respectively. The prediction times of the proposed approach, *SVR*, *FSVR*, and *DTR* are 0.04 minutes, 0.0005 minutes, 0.0003 minutes, and 0.0006 minutes, respectively.

5. Threats to Validity

A threat to the construct validity of this study involves the selection of the performance criteria (i.e., *MSE* and r^2) for evaluating the defect prediction model. We adopted these metrics because they have been widely used in previous studies [18] [101].

A threat to the internal validity of this study involves the replication of the competing approaches and the parameters of the defect prediction models. Internal validity is concerned with various uncontrolled internal factors that may affect the results. To mitigate this threat, we used third-party libraries, such as packages from scikit-learn, and double checked the implementation of the competing approaches.

The first threat to external validity involves dataset selection. The different metrics used for the different datasets may affect the defect prediction performance, which in turn may affect the performance of the proposed approach on other datasets.

The second threat to external validity involves the generalizability of the proposed approach. We evaluated our proposed approach on only two pub-

licly available datasets. Thus, the results may not be generalizable to other software projects, such as commercial projects.

6. Conclusions and Future Work

In this paper, we proposed a deep learning-based approach to predict the number of defects in software modules. Our proposed approach trains a deep learning model to predict the number of defects. On the given the datasets, the performance improvement of the proposed approach upon the *SVR*, *FSVR*, and *DTR* is significant. Compared with these state-of-art approaches on two well-known datasets, the proposed approach achieves a significant reduction in the mean square error (varying between 3% and 13%) and improves the squared correlation coefficient (varying between 2% and 27%).

In future work, we plan to investigate the number of defect predictions in software modules by including more projects written in different programming languages and commercial projects from industry. We are also interested in predicting the number of defects from the change level rather than from the module level.

In cooperating with software testing engineers to construct new software complexity metric would be another interesting research task. In the defect prediction field, additional new and commercial datasets are needed. It would also be interesting to investigate new performance criteria for defect prediction models. Finally, we plan to investigate effort-aware defect prediction models and attempt to determine the agent of the effort through an effort-aware defect prediction model and the relations between the effort and the agent.

Declaration of interests

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

The research work described in this paper was partially supported by the National Natural Science Foundation of China (61772071) and the grant

from the Joint Research Fund in Astronomy (U1531242) under cooperative agreement between the NSFC and CAS.

References

[1] D. Bowes, T. Hall, J. Petrić, Software defect prediction: do different classifiers find the same defects?, Software Quality Journal 26 (2) (2018) 525–552. doi:10.1007/s11219-016-9353-3.
URL https://doi.org/10.1007/s11219-016-9353-3

[2] N. E. Fenton, M. Neil, A critique of software defect prediction models, IEEE Transactions on Software Engineering 25 (5) (1999) 675–689. doi:10.1109/32.815326.

[3] V. A. Prakash, D. V. Ashoka, V. N. M. Aradya, Application of data mining techniques for defect detection and classification, in: S. C. Satapathy, B. N. Biswal, S. K. Udgata, J. Mandal (Eds.), Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014, Springer International Publishing, Cham, 2015, pp. 387–395.

[4] G. Tassey, The economic impacts of inadequate infrastructure for software testing (2002).

[5] J. Nam, W. Fu, S. Kim, T. Menzies, L. Tan, Heterogeneous defect prediction, IEEE Transactions on Software Engineering 44 (9) (2018) 874–896. doi:10.1109/TSE.2017.2720603.
URL doi.ieeecomputersociety.org/10.1109/TSE.2017.2720603

[6] K. E. Bennin, J. Keung, P. Phannachitta, A. Monden, S. Mensah, Mahakil: Diversity based oversampling approach to alleviate the class imbalance issue in software defect prediction, IEEE Transactions on Software Engineering 44 (6) (2018) 534–550. doi:10.1109/TSE.2017.2731766.

[7] M. Yan, Y. Fang, D. Lo, X. Xia, X. Zhang, File-level defect prediction: Unsupervised vs. supervised models, in: Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '17, IEEE Press, Piscataway, NJ, USA, 2017, pp. 344–353. doi:10.1109/ESEM.2017.48.
URL https://doi.org/10.1109/ESEM.2017.48

[8] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, A. E. Hassan, Studying just-in-time defect prediction using cross-project models, Empirical Software Engineering 21 (5) (2016) 2072–2106. doi:10.1007/s10664-015-9400-x.
URL https://doi.org/10.1007/s10664-015-9400-x

[9] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, N. Ubayashi, A large-scale empirical study of just-in-time quality assurance, IEEE Transactions on Software Engineering 39 (6) (2013) 757–773. doi:10.1109/TSE.2012.70.

[10] X.-Y. Jing, S. Ying, Z.-W. Zhang, S.-S. Wu, J. Liu, Dictionary learning based software defect prediction, in: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, ACM, New York, NY, USA, 2014, pp. 414–423. doi:10.1145/2568225.2568320.
URL http://doi.acm.org/10.1145/2568225.2568320

[11] M. Tan, L. Tan, S. Dara, C. Mayeux, Online defect prediction for imbalanced data, in: Proceedings of the 37th International Conference on Software Engineering - Volume 2, ICSE '15, IEEE Press, Piscataway, NJ, USA, 2015, pp. 99–108.
URL http://dl.acm.org/citation.cfm?id=2819009.2819026

[12] A. Schröter, T. Zimmermann, A. Zeller, Predicting component failures at design time, in: Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering, ISESE '06, ACM, New York, NY, USA, 2006, pp. 18–27. doi:10.1145/1159733.1159739.
URL http://doi.acm.org/10.1145/1159733.1159739

[13] H. Hata, O. Mizuno, T. Kikuno, Bug prediction based on fine-grained module histories, in: Proceedings of the 34th International Conference on Software Engineering, ICSE '12, IEEE Press, Piscataway, NJ, USA, 2012, pp. 200–210.
URL http://dl.acm.org/citation.cfm?id=2337223.2337247

[14] H. Tang, T. Lan, D. Hao, L. Zhang, Enhancing defect prediction with static defect analysis, in: Proceedings of the 7th Asia-Pacific Symposium on Internetware, Internetware '15, ACM, New York, NY, USA, 2015, pp. 43–51. doi:10.1145/2875913.2875922.
URL http://doi.acm.org/10.1145/2875913.2875922

[15] M. H. Halstead, Elements of Software Science (Operating and Programming Systems Series), Elsevier Science Inc., New York, NY, USA, 1977.

[16] T. J. McCabe, A complexity measure, in: Proceedings of the 2Nd International Conference on Software Engineering, ICSE '76, IEEE Computer Society Press, Los Alamitos, CA, USA, 1976, pp. 407–.
URL http://dl.acm.org/citation.cfm?id=800253.807712

[17] F. Xing, P. Guo, Support vector regression for software reliability growth modeling and prediction, in: J. Wang, X. Liao, Z. Yi (Eds.), Advances in Neural Networks – ISNN 2005, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 925–930.

[18] Z. Yan, X. Chen, P. Guo, Software defect prediction using fuzzy support vector regression, in: L. Zhang, B.-L. Lu, J. Kwok (Eds.), Advances in Neural Networks - ISNN 2010, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 17–24.

[19] K. E. Bennin, K. Toda, Y. Kamei, J. Keung, A. Monden, N. Ubayashi, Empirical evaluation of cross-release effort-aware defect prediction models, in: 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS), 2016, pp. 214–221. doi:10.1109/QRS.2016.33.

[20] S. Amasaki, Cross-version defect prediction using cross-project defect prediction approaches: Does it work?, in: Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE'18, ACM, New York, NY, USA, 2018, pp. 32–41. doi:10.1145/3273934.3273938.
URL http://doi.acm.org/10.1145/3273934.3273938

[21] S. Lessmann, B. Baesens, C. Mues, S. Pietsch, Benchmarking classification models for software defect prediction: A proposed framework and novel findings, IEEE Transactions on Software Engineering 34 (4) (2008) 485–496. doi:10.1109/TSE.2008.35.

[22] T. Jiang, L. Tan, S. Kim, Personalized defect prediction, in: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2013, pp. 279–289. doi:10.1109/ASE.2013.6693087.

[23] T. Menzies, A. Butcher, D. Cok, A. Marcus, L. Layman, F. Shull, B. Turhan, T. Zimmermann, Local versus global lessons for defect prediction and effort estimation, IEEE Transactions on Software Engineering 39 (6) (2013) 822–834. doi:10.1109/TSE.2012.83.

[24] E. Kocaguneli, T. Menzies, A. Bener, J. W. Keung, Exploiting the essential assumptions of analogy-based effort estimation, IEEE Transactions on Software Engineering 38 (2) (2012) 425–438. doi:10.1109/TSE.2011.27.

[25] B. Ghotra, S. McIntosh, A. E. Hassan, 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, IEEE Press, Piscataway, NJ, USA, 2015, pp. 789–800.
URL http://dl.acm.org/citation.cfm?id=2818754.2818850

[26] S. S. Rathore, S. Kumar, An empirical study of some software fault prediction techniques for the number of faults prediction, Soft Comput. 21 (24) (2017) 7417–7434. doi:10.1007/s00500-016-2284-x.
URL https://doi.org/10.1007/s00500-016-2284-x

[27] T. J. Ostrand, E. J. Weyuker, R. M. Bell, Predicting the location and number of faults in large software systems, IEEE Transactions on Software Engineering 31 (4) (2005) 340–355. doi:10.1109/TSE.2005.49.

[28] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, A. Ihara, K. Matsumoto, The impact of mislabelling on the performance and interpretation of defect prediction models, in: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 1, 2015, pp. 812–823. doi:10.1109/ICSE.2015.93.

[29] X. Yu, J. Liu, Z. Yang, X. Jia, Q. Ling, S. Ye, Learning from imbalanced data for predicting the number of software defects, in: 2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE), IEEE Computer Society, Los Alamitos, CA, USA, 2017, pp. 78–89. doi:10.1109/ISSRE.2017.18.
URL https://doi.ieeecomputersociety.org/10.1109/ISSRE.2017.18

[30] S. S. Rathore, S. Kumar, A decision tree regression based approach for the number of software faults prediction, SIGSOFT Softw. Eng. Notes 41 (1) (2016) 1–6. doi:10.1145/2853073.2853083.
URL http://doi.acm.org/10.1145/2853073.2853083

[31] M. Chen, Y. Ma, An empirical study on predicting defect numbers, in: SEKE'15, 2015, pp. 397–402. doi:10.18293/SEKE2015-132.

[32] Y. Qian, P. C. Woodland, Very deep convolutional neural networks for robust speech recognition, CoRR abs/1610.00277. arXiv:1610.00277.
URL http://arxiv.org/abs/1610.00277

[33] T. Young, D. Hazarika, S. Poria, E. Cambria, Recent trends in deep learning based natural language processing, CoRR abs/1708.02709. arXiv:1708.02709.
URL http://arxiv.org/abs/1708.02709

[34] T. Goswami, Impact of deep learning in image processing and computer vision, in: J. Anguera, S. C. Satapathy, V. Bhateja, K. Sunitha (Eds.), Microelectronics, Electromagnetics and Telecommunications, Springer Singapore, Singapore, 2018, pp. 475–485.

[35] Z. Wan, X. Xia, A. E. Hassan, D. Lo, J. Yin, X. Yang, Perceptions, expectations, and challenges in defect prediction, IEEE Transactions on Software Engineering (2018) 1–1doi:10.1109/TSE.2018.2877678.

[36] Mei-Huei Tang, Ming-Hung Kao, Mei-Hwa Chen, An empirical study on object-oriented metrics, in: Proceedings Sixth International Software Metrics Symposium (Cat. No.PR00403), 1999, pp. 242–249. doi:10.1109/METRIC.1999.809745.

[37] S. R. Chidamber, C. F. Kemerer, A metrics suite for object oriented design, IEEE Transactions on Software Engineering 20 (6) (1994) 476–493. doi:10.1109/32.295895.

[38] T. Yu, W. Wen, X. Han, J. Hayes, Conpredictor: Concurrency defect prediction in real-world applications, IEEE Transactions on Software Engineering (2018) 1–1doi:10.1109/TSE.2018.2791521.

[39] T. Zimmermann, R. Premraj, A. Zeller, Predicting defects for eclipse, in: Third International Workshop on Predictor Models in Software

Engineering (PROMISE'07: ICSE Workshops 2007), 2007, pp. 9–9. doi:10.1109/PROMISE.2007.10.

[40] C. Catal, B. Diri, A systematic review of software fault prediction studies, Expert Syst. Appl. 36 (4) (2009) 7346–7354. doi:10.1016/j.eswa.2008.10.027.
URL http://dx.doi.org/10.1016/j.eswa.2008.10.027

[41] M. D'Ambros, M. Lanza, R. Robbes, Evaluating defect prediction approaches: a benchmark and an extensive comparison, Empirical Software Engineering 17 (4) (2012) 531–577. doi:10.1007/s10664-011-9173-9.
URL https://doi.org/10.1007/s10664-011-9173-9

[42] J. Nam, S. Kim, Clami: Defect prediction on unlabeled datasets (t), in: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2015, pp. 452–463. doi:10.1109/ASE.2015.56.

[43] R. Ozakinci, A. Tarhan, The role of process in early software defect prediction: Methods, attributes and metrics, in: P. M. Clarke, R. V. O'Connor, T. Rout, A. Dorling (Eds.), Software Process Improvement and Capability Determination, Springer International Publishing, Cham, 2016, pp. 287–300.

[44] F. Zhang, Q. Zheng, Y. Zou, A. E. Hassan, Cross-project defect prediction using a connectivity-based unsupervised classifier, in: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), 2016, pp. 309–320. doi:10.1145/2884781.2884839.

[45] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, H. Leung, 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, ACM, New York, NY, USA, 2016, pp. 157–168. doi:10.1145/2950290.2950353.
URL http://doi.acm.org/10.1145/2950290.2950353

[46] P. Guo, M. Lyu, A pseudoinverse learning algorithm for feedforward neural networks with stacked generalization applications to software reliability growth data, Neurocomputing 56 (1) (2004) 101–121.

[47] F. Xing, P. Guo, M. R. Lyu, A novel method for early software quality prediction based on support vector machine, in: 16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05), 2005, pp. 10 pp.–222. doi:10.1109/ISSRE.2005.6.

[48] S. Zhong, T. M. Khoshgoftaar, N. Seliya, Unsupervised learning for expert-based software quality estimation, in: Eighth IEEE International Symposium on High Assurance Systems Engineering, 2004. Proceedings., 2004, pp. 149–155. doi:10.1109/HASE.2004.1281739.

[49] S. Herbold, A. Trautsch, J. Grabowski, A comparative study to benchmark cross-project defect prediction approaches, IEEE Transactions on Software Engineering 44 (9) (2018) 811–833. doi:10.1109/TSE.2017.2724538.

[50] A. E. C. Cruz, K. Ochimizu, Towards logistic regression models for predicting fault-prone code across software projects, in: 2009 3rd International Symposium on Empirical Software Engineering and Measurement, 2009, pp. 460–463. doi:10.1109/ESEM.2009.5316002.

[51] I. H. Laradji, M. Alshayeb, L. Ghouti, Software defect prediction using ensemble learning on selected features, Information and Software Technology 58 (2015) 388–402. doi:{10.1016/j.infsof.2014.07.005}.

[52] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, K. Matsumoto, Automated parameter optimization of classification techniques for defect prediction models, in: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), 2016, pp. 321–332. doi:10.1145/2884781.2884857.

[53] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, K. Matsumoto, The impact of automated parameter optimization on defect prediction models, IEEE Transactions on Software Engineering (2018) 1–1doi:10.1109/TSE.2018.2794977.

[54] B. Ghotra, S. Mcintosh, A. E. Hassan, A large-scale study of the impact of feature selection techniques on defect classification models, in: Proceedings of the 14th International Conference on Mining Software Repositories, MSR '17, IEEE Press, Piscataway, NJ, USA, 2017, pp.

146–157. doi:10.1109/MSR.2017.18.
URL https://doi.org/10.1109/MSR.2017.18

[55] P. He, B. Li, X. Liu, J. Chen, Y. Ma, An empirical study on software defect prediction with a simplified metric set, Inf. Softw. Technol. 59 (C) (2015) 170–190. doi:10.1016/j.infsof.2014.11.006.
URL http://dx.doi.org/10.1016/j.infsof.2014.11.006

[56] Y. Zhang, D. Lo, X. Xia, J. Sun, Combined classifier for cross-project defect prediction: an extended empirical study, Frontiers of Computer Science 12 (2) (2018) 280–296. doi:10.1007/s11704-017-6015-y.
URL https://doi.org/10.1007/s11704-017-6015-y

[57] A. Mockus, D. M. Weiss, Predicting risk of software changes, Bell Labs Technical Journal 5 (2) (2000) 169–180. doi:10.1002/bltj.2229.

[58] N. Ohlsson, H. Alberg, Predicting fault-prone software modules in telephone switches, IEEE Transactions on Software Engineering 22 (12) (1996) 886–894. doi:10.1109/32.553637.

[59] X. Yang, K. Tang, X. Yao, A learning-to-rank approach to software defect prediction, IEEE Transactions on Reliability 64 (1) (2015) 234–246. doi:10.1109/TR.2014.2370891.

[60] E. A. Felix, S. P. Lee, Integrated approach to software defect prediction, IEEE Access 5 (2017) 21524–21547. doi:10.1109/ACCESS.2017.2759180.

[61] V. Vapnik, S. E. Golowich, A. J. Smola, Support vector method for function approximation, regression estimation and signal processing, in: M. C. Mozer, M. I. Jordan, T. Petsche (Eds.), Advances in Neural Information Processing Systems 9, MIT Press, 1997, pp. 281–287.
URL http://papers.nips.cc/paper/1187-support-vector-method-for-function-approximation-regression-esti\mation-and-signal-processing.pdf

[62] V. N. Vapnik, The nature of statistical learning theory, Springer-Verlag New York, Inc., New York, NY, USA, 1995.

[63] H. Drucker, C. J. C. Burges, L. Kaufman, A. J. Smola, V. Vapnik, Support vector regression machines, in: M. C. Mozer, M. I. Jordan,

T. Petsche (Eds.), Advances in Neural Information Processing Systems 9, MIT Press, 1997, pp. 155–161.
URL http://papers.nips.cc/paper/1238-support-vector-regression-machines.pdf

[64] S. W. Lee, D. S. Kim, M. G. Na, Prediction of dnbr using fuzzy support vector regression and uncertainty analysis, IEEE Transactions on Nuclear Science 57 (3) (2010) 1595–1601. doi:10.1109/TNS.2010.2047265.

[65] T. Le, D. Tran, W. Ma, D. Sharma, A new fuzzy membership computation method for fuzzy support vector machines, in: International Conference on Communications and Electronics 2010, 2010, pp. 153–157. doi:10.1109/ICCE.2010.5670701.

[66] Y. Zhang, S. Xu, K. Chen, Z. Liu, C. P. Chen, Fuzzy density weight-based support vector regression for image denoising, Information Sciences 339 (2016) 175 – 188. doi:https://doi.org/10.1016/j.ins.2016.01.007.
URL http://www.sciencedirect.com/science/article/pii/S0020025516000098

[67] Zonghai Sun, Youxian Sun, Fuzzy support vector machine for regression estimation, in: SMC'03 Conference Proceedings. 2003 IEEE International Conference on Systems, Man and Cybernetics. Conference Theme - System Security and Assurance (Cat. No.03CH37483), Vol. 4, 2003, pp. 3336–3341 vol.4. doi:10.1109/ICSMC.2003.1244404.

[68] Z. Liu, S. Xu, C. L. P. Chen, Y. Zhang, X. Chen, Y. Wang, A three-domain fuzzy support vector regression for image denoising and experimental studies, IEEE Transactions on Cybernetics 44 (4) (2014) 516–525. doi:10.1109/TSMCC.2013.2258337.

[69] L. Chen, M. Zhou, M. Wu, J. She, Z. Liu, F. Dong, K. Hirota, Three-layer weighted fuzzy support vector regression for emotional intention understanding in humanCrobot interaction, IEEE Transactions on Fuzzy Systems 26 (5) (2018) 2524–2538. doi:10.1109/TFUZZ.2018.2809691.

[70] J. R. Quinlan, Induction of decision trees, Machine Learning 1 (1) (1986) 81–106. doi:10.1007/BF00116251.
URL https://doi.org/10.1007/BF00116251

[71] S. Naeem, T. M.Khoshgoftaar, The use of decision trees for cost-sensitive classification: an empirical study in software quality prediction, Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery 1 (5) (2011) 448–459. doi:10.1002/widm.38.

[72] A. Krizhevsky, I. Sutskever, G. E. Hinton, Imagenet classification with deep convolutional neural networks, Commun. ACM 60 (6) (2017) 84–90. doi:10.1145/3065386.
URL http://doi.acm.org/10.1145/3065386

[73] J. L. Elman, Finding structure in time, Cognitive Science 14 (2) (1990) 179–211. doi:https://doi.org/10.1016/0364-0213(90)90002-E.
URL http://www.sciencedirect.com/science/article/pii/036402139090002E

[74] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural Comput. 9 (8) (1997) 1735–1780. doi:10.1162/neco.1997.9.8.1735.
URL http://dx.doi.org/10.1162/neco.1997.9.8.1735

[75] G. Zhao, J. Huang, Deepsim: Deep learning code functional similarity, in: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018, ACM, New York, NY, USA, 2018, pp. 141–151. doi:10.1145/3236024.3236068.
URL http://doi.acm.org/10.1145/3236024.3236068

[76] S. Ma, Y. Liu, W.-C. Lee, X. Zhang, A. Grama, Mode: Automated neural network model debugging via state differential analysis and input selection, in: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018, ACM, New York, NY, USA, 2018, pp. 175–186. doi:10.1145/3236024.3236082.
URL http://doi.acm.org/10.1145/3236024.3236082

[77] J. Guo, J. Cheng, J. Cleland-Huang, Semantically enhanced software traceability using deep learning techniques, in: 2017 IEEE/ACM 39th

International Conference on Software Engineering (ICSE), 2017, pp. 3–14. doi:10.1109/ICSE.2017.9.

[78] M. White, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, Toward deep learning software repositories, in: Proceedings of the 12th Working Conference on Mining Software Repositories, MSR '15, IEEE Press, Piscataway, NJ, USA, 2015, pp. 334–345.
URL http://dl.acm.org/citation.cfm?id=2820518.2820559

[79] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, D. Tarlow, Deepcoder: Learning to write programs, in: 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings, 2017.
URL https://openreview.net/forum?id=ByldLrqlx

[80] Y. Tian, K. Pei, S. Jana, B. Ray, Deeptest: Automated testing of deep-neural-network-driven autonomous cars, in: Proceedings of the 40th International Conference on Software Engineering, ICSE '18, ACM, New York, NY, USA, 2018, pp. 303–314. doi:10.1145/3180155.3180220.
URL http://doi.acm.org/10.1145/3180155.3180220

[81] X. Huo, M. Li, Z.-H. Zhou, Learning unified features from natural and programming languages for locating buggy source code, in: Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI'16, AAAI Press, 2016, pp. 1606–1612.
URL http://dl.acm.org/citation.cfm?id=3060832.3060845

[82] X. Gu, H. Zhang, D. Zhang, S. Kim, Deep api learning, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, ACM, New York, NY, USA, 2016, pp. 631–642. doi:10.1145/2950290.2950334.
URL http://doi.acm.org/10.1145/2950290.2950334

[83] M. White, M. Tufano, C. Vendome, D. Poshyvanyk, Deep learning code fragments for code clone detection, in: 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), 2016, pp. 87–98.

[84] B. Xu, D. Ye, Z. Xing, X. Xia, G. Chen, S. Li, Predicting semantically linkable knowledge in developer online forums via convolutional

neural network, in: 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), 2016, pp. 51–62.

[85] A. N. Lam, A. T. Nguyen, H. A. Nguyen, T. N. Nguyen, 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, pp. 476–481. doi:10.1109/ASE.2015.73.

[86] L. Mou, G. Li, L. Zhang, T. Wang, Z. Jin, 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, pp. 1287–1293.
URL http://dl.acm.org/citation.cfm?id=3015812.3016002

[87] X. Gu, H. Zhang, S. Kim, Deep code search, in: Proceedings of the 40th International Conference on Software Engineering, ICSE '18, ACM, New York, NY, USA, 2018, pp. 933–944. doi:10.1145/3180155.3180167.
URL http://doi.acm.org/10.1145/3180155.3180167

[88] V. J. Hellendoorn, C. Bird, E. T. Barr, M. Allamanis, Deep learning type inference, in: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018, ACM, New York, NY, USA, 2018, pp. 152–162. doi:10.1145/3236024.3236051.
URL http://doi.acm.org/10.1145/3236024.3236051

[89] J. Henkel, S. K. Lahiri, B. Liblit, T. Reps, Code vectors: Understanding programs through embedded abstracted symbolic traces, in: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018, ACM, New York, NY, USA, 2018, pp. 163–174. doi:10.1145/3236024.3236085.
URL http://doi.acm.org/10.1145/3236024.3236085

[90] Q. Huang, X. Xia, D. Lo, Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction, in: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2017, pp. 159–170. doi:10.1109/ICSME.2017.51.

[91] F. R. Porto, L. L. Minku, E. Mendes, A. Simão, A systematic study of cross-project defect prediction with meta-learning, CoRR abs/1802.06025. arXiv:1802.06025.
URL http://arxiv.org/abs/1802.06025

[92] C. FENG, H. WANG, N. LU, T. CHEN, H. HE, Y. LU, X. M. TU, Log-transformation and its implications for data analysis, Shanghai Arch Psychiatry 26. doi:10.3969/j.issn.1002-0829.2014.02.009.
URL http://www.ncbi.nlm.nih.gov/pmc/articles/PMC4120293/

[93] S. G. K. Patro, K. K. Sahu, Normalization: A preprocessing stage, CoRR abs/1503.06462. arXiv:1503.06462.
URL http://arxiv.org/abs/1503.06462

[94] P. Jiaqi, Z. Yan, F. Simon, 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, pp. 72–78. doi:10.1007/978-981-10-2777-2.

[95] J. Nam, S. J. Pan, S. Kim, Transfer defect learning, in: 2013 35th International Conference on Software Engineering (ICSE), 2013, pp. 382–391. doi:10.1109/ICSE.2013.6606584.

[96] V. Gajera, Shubham, R. Gupta, P. K. Jana, An effective multi-objective task scheduling algorithm using min-max normalization in cloud computing, in: 2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT), 2016, pp. 812–816. doi:10.1109/ICATCCT.2016.7912111.

[97] Y. K. Jain, S. K. Bhandare, Min max normalization based data perturbation method for privacy protection, International Journal of Computer and Communication Technology 2.

[98] S. Bhanja, A. Das, Impact of data normalization on deep neural network for time series forecasting, CoRR abs/1812.05519. arXiv:1812.05519.
URL http://arxiv.org/abs/1812.05519

[99] P. Sane, R. Agrawal, Pixel normalization from numeric data as input to neural networks: For machine learning and image processing, in: 2017 International Conference on Wireless Communications,

Signal Processing and Networking (WiSPNET), 2017, pp. 2221–2225. doi:10.1109/WiSPNET.2017.8300154.

[100] J. Sola, J. Sevilla, Importance of input data normalization for the application of neural networks to complex industrial problems, IEEE Transactions on Nuclear Science 44 (3) (1997) 1464–1468. doi:10.1109/23.589532.

[101] X. Jin, Z. Liu, R. Bie, G. Zhao, J. Ma, Support vector machines for regression and applications to software quality prediction, in: V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, J. Dongarra (Eds.), Computational Science – ICCS 2006, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 781–788.

[102] M. R. Lyu (Ed.), Handbook of Software Reliability Engineering, McGraw-Hill, Inc., Hightstown, NJ, USA, 1996.

[103] S. Dick, A. Meeks, M. Last, H. Bunke, A. Kandel, Data mining in software metrics databases, Fuzzy Sets and Systems 145 (1) (2004) 81 – 110, computational Intelligence in Software Engineering. doi:https://doi.org/10.1016/j.fss.2003.10.006.
URL http://www.sciencedirect.com/science/article/pii/S016501140300438X

[104] Promise software engineering repository.
URL http://promise.site.uottawa.ca/SERepository/datasets-page.html

[105] M. Torabi, J. Rao, Estimation of mean squared error of model-based estimators of small area means under a nested error linear regression model, Journal of Multivariate Analysis 117 (2013) 76 – 87. doi:https://doi.org/10.1016/j.jmva.2013.02.008.
URL http://www.sciencedirect.com/science/article/pii/S0047259X13000225

[106] Kvalseth, T. O., Cautionary note about r2, The American Statistician 39 (4) (1985) 279–285.

[107] Z. Wang, A. C. Bovik, Mean squared error: Love it or leave it? a new look at signal fidelity measures, IEEE Signal Processing Magazine 26 (1) (2009) 98–117. doi:10.1109/MSP.2008.930649.

[108] J. E. Figueroa-Lpez, C. Mancini, Optimum thresholding using mean and conditional mean squared error, Journal of Econometrics 208 (1) (2019) 179 – 210. doi:https://doi.org/10.1016/j.jeconom.2018.09.011.
URL http://www.jstor.org/stable/24310360

[109] R. C. Steorts, M. Ghosh, On estimation of mean squared errors of benchmarked empirical bayes estimators, Statistica Sinica 23 (2) (2013) 757–773.
URL http://www.jstor.org/stable/24310360

[110] R. Niu, J. Lu, False information detection with minimum mean squared errors for bayesian estimation, in: 2015 49th Annual Conference on Information Sciences and Systems (CISS), 2015, pp. 1–6. doi:10.1109/CISS.2015.7086893.

[111] R. Hawkes, P. Date, A statistical test for the mean squared error, Journal of Statistical Computation and Simulation 63 (4) (1999) 321–347. doi:doi:10.1080/00949659908811960.
URL https://doi.org/10.1080/00949659908811960

[112] D. L. J. Alexander, A. Tropsha, D. A. Winkler, Beware of r2: Simple, unambiguous assessment of the prediction accuracy of qsar and qspr models, Journal of chemical information and modeling 55 (7) (2015) 1316–22. doi:10.1021/acs.jcim.5b00206.

[113] A. Schneider, G. Hommel, M. Blettner, Linear regression analysis: part 14 of a series on evaluation of scientific publications, Deutsches Arzteblatt international 107 (44) (2010) 776–82. doi:10.3238/arztebl.2010.0776.

[114] N. E. Fenton, N. Ohlsson, Quantitative analysis of faults and failures in a complex software system, IEEE Transactions on Software Engineering 26 (8) (2000) 797–814. doi:10.1109/32.879815.

[115] A. G. Koru, D. Zhang, K. El Emam, H. Liu, An investigation into the functional form of the size-defect relationship for software modules, IEEE Transactions on Software Engineering 35 (2) (2009) 293–304. doi:10.1109/TSE.2008.90.

[116] T. Galinac Grbac, P. Runeson, D. Huljeni?, A second replicated quantitative analysis of fault distributions in complex software systems, IEEE Transactions on Software Engineering 39 (4) (2013) 462–476. doi:10.1109/TSE.2012.46.

Lei Qiao received B.S. and M.S. degrees in computer science and technology from North China University of Technology, Beijing, China. He is currently pursuing a Ph.D. degree in computer science and technology at Beijing Institute of Technology, Beijing, China. His research interests include defect prediction, software testing, machine learning and neural networks in software engineering.

Xuesong Li received his Ph.D. degree from the Center for Biomedical Imaging Research, Tsinghua University, and he is an assistant professor at Beijing Institute of Technology. His academic and clinical focus is the use of algorithms (deep learning, machine learning, big data analysis, etc.), advanced medical imaging techniques (sMRI, DTI, fMRI, etc.) and software defect prediction technology to improve clinical practice for neurological diseases.

Qasim Umer received a B.S. degree in computer science from Punjab University, Pakistan in 2006, an M.S. degree in .net distributed system development from University of Hull, UK in 2009, and an M.S. degree in computer science from University of Hull, UK in 2012. He is currently pursuing a Ph.D. degree in computer science from Beijing Institute of Technology, China. He is particularly interested in machine learning, data mining and software maintenance.

Dr. Ping Guo currently is a professor at the school of systems science in Beijing Normal University, the founding Director of the Key Laboratory of graphics, image and pattern recognition at Beijing Normal University. He is the IEEE senior member and CCF senior member. His research interests include computational intelligence theory as well as applications in pattern recognition, image processing, software reliability engineering, and astronomical data processing. He has published more than 360 papers, hold 6 patents, and is the author of two Chinese

books:"*Computational intelligence in software reliability engineering*", and "*Image semantic analysis*". He received 2012 Beijing municipal government award of science and technology (third rank) entitled "*regularization method and its application*".

Professor Guo received his master's degree in optics from the Department of physics, Peking University, and received his PhD degree in computer science from the Department of computer science and engineering, Chinese University Hong Kong.