Received 27 September 2022, accepted 21 October 2022, date of publication 26 October 2022, date of current version 7 November 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3217480

RESEARCH ARTICLE

Software Defect Density Prediction Using Deep Learning

FIRAS ALGHANIM, (Member, IEEE), MOHAMMAD AZZEH, AMMAR EL-HASSAN, AND HAZEM QATTOUS
King Hussein School of Computing Sciences, Princess Sumaya University for Technology, Amman 11941, Jordan
Corresponding author: Firas Alghanim (f.ghanim@psut.edu.jo)

ABSTRACT Delivering a reliable and high-quality software system to client is a big challenge in software development and evolution process. One of the software measures that confirm the quality of the system is the defect density. Practitioners usually need this measure during software development process or during a period of operation to indicate the reliability of software system. However, since predicting defect density before testing the modules is time consuming, managers need to build a prediction model that can help in detecting the defective modules. This process can reduce the testing cost and improve testing resources utilizations. The most intrinsic feature of software defect datasets is the data sparsity in the defect density which might bias the final prediction. Therefore, we use deep learning to build defect density prediction models and handle the inherit challenge of data sparsity in defect density. Deep learning has shown to be effective with sparse data. The constructed model has been evaluated against well-known machine learning methods over 28 public datasets. The obtained results confirmed that the deep learning model is generally more adequate than other machine models over the datasets with high and very high sparsity ratios, and competitive choice when the sparsity ratio is either medium or low.

INDEX TERMS Defect density prediction, deep learning, data sparsity, machine learning.

I. INTRODUCTION

Quality and reliability of software products are important indicators for the success of software development process [1], [2]. Software engineers always apply different testing and quality assurance activities to ensure that the software product meets the quality standard before delivering it to the customers [3], [4], [5], [6]. Defect Density is a popular software metric that is used to confirm the quality of software product, especially during software evolution [7]. It is defined as number of defects per software module[1] size during a period of operation or during software development, as show in equation (1). Predicting Defect Density early can afford several advantages to the quality assurance team [8], [9]. First, it can optimize resources utilization for the limited budget projects [7], [10]. Software engineers often use the predicted defected density to rank and priorities software modules to pay more attention to the software modules that are defects prone [11]. Second, it measures the effectiveness of testing process and third, it is useful in pointing towards high-risk components [8], [12], [13].

$$DD = \frac{bug_{count}}{LOC} \times 1000 \quad (1)$$

The prediction of defect density is conducted by extracting static code metrics from bug log files of prior software versions. These static metrics are used to construct models that predict the potential defect density in other modules or future releases of the same module [12], [13]. This approach assists in locating the areas of the software that are prone to cause errors. Defect density prediction can be applied in two ways: within the project and across the projects. The first method entails utilizing the same data for both testing and training during the empirical validation procedure [6], [14]. In the second method, one release of project data serves as training, while the subsequent release serves as testing [15].

The associate editor coordinating the review of this manuscript and approving it for publication was Ramesh Babu N.

[1]Software module, either a routine or a class, is usually sized by Line of Code (LOC).

Predicting Defect density is different than defect prediction even though both approaches can use the same dataset features. The major task of defect prediction is to identify whether the module in hand is defected or not (i.e., classification task), whereas in defect density prediction, we are interested in predicting the density of defects per module size (i.e., regression task). Defect density prediction has not received substantially enough attention from researchers in comparison to defect prediction, because not all datasets contain bug counts that facilitates computing defect density in the training datasets. However, the majority of studies on defect density applied statistical regression and machine learning techniques, and a few of them focus on analyzing statistical properties between defect density and static code metrics [5].

The main challenge that affects the success of defect density prediction is the presence of sparsity in the defect density variable. Software defect datasets are frequently dense, but the target variable (i.e., defect density) is often sparse. Sparsity means that the majority of the observations have zero value in the defect density variable due to the absence of bugs in the majority of training modules. Naturally, most of the employed defect datasets share the same sparsity problem, and the traditional machine learning problem cannot deal effectively with this challenge. Existing machine learning models struggle to capture complex relationships between input dense data and sparse defect density variable. Therefore, we use Deep Neural Networks (DNN) as potential solution for this problem. Unlike other learning algorithms, the deep learning has shown to be effective to learn from a large and dense data. In addition to that, it finds a complex patterns in the data as in our case when defect density variable is sparse. Deep learning has been widely used in a variety of research areas such as natural language processing, speech recognition, and image processing [16].

Mainly, we have enhanced a well-known artificial neural network called Generalized Regression Neural Networks (GRNN) with multiple dense layers to fit with the defect density predictions as explained in section 4. GRNN is a single-pass neural network which uses a Gaussian activation function in the hidden layer and often used for function approximation. GRNN consists of input, hidden, summation, and division layers. Since the defect density variable is sparse, learning the relationship between input variables and target variables will be affected by the bias in the target variable. Therefore, using GRNN model can alleviate this problem by involving the distribution of the data in the learning process. The GRNN was employed in this study for three reasons: 1) the GRNN does not require backpropagation for training the network, thus reducing computational time and the storage required. 2) the GRNN shows high accuracy in the estimation since it uses Gaussian functions. 3) the GRNN can handle noises in the data.

In this study we propose a deep learning model to discover complex patterns in the defect datasets and improve the accuracy of defect density prediction. Also, it can learn and capture the discriminative features from data automatically, thus resulting in a more accurate defect density prediction model [17]. To accomplish that, We extended the traditional GRNN by additional three hidden deep layers. The proposed model is an extension of the conventional GRNN to solve the sparsity problem in the defect density predictions. Interestingly, deep learning has not used before to predict defect density, it was used intensively for defect prediction as classification algorithm.

To examine the effectiveness of deep learning with data sparsity in defect density prediction, we divided datasets into four homogeneous groups based on the level of sparsity. The contributions of this paper are threefold: 1) extend GRNN with multiple hidden layers to improve accuracy of the defect density prediction, and 2) alleviate the problem of sparsity in defect density, and 3) we predict defect density at the module level, not the project level which help quality assurance to better monitor the project progress The proposed work has been driven by the following research questions:

RQ1: *Does Deep learning outperform the traditional machine learning algorithms for defect density?*

To address this question, we built an enhanced version of traditional GRNN with multiple dense layers. Then, we compare it with other common machine learning algorithms such as k-Nearest Neighbors (kNN), Support Vector Regression (SVR), Random Forest (RF), Naïve Bayes (NB), and Multiple Linear Regression (MLR) based on 28 public datasets. We used the repeated 10-Folds cross-validation to validate the constructed model and other machine learning models. The results are statistically compared using Wilcoxon signed-rank test, which offer evidence that the proposed deep GRNN model is significantly better than other comparative machine learning models.

RQ2: *What is the role of data sparsity in defect density prediction based deep learning?*

This question addresses the role of data sparsity in defect density predictions. The sparsity ratio, as defined in equation (2), measures the percentage of non-zero defected modules (minority) to the zero defected modules (majority). Based on the SR, we divided the datasets into four groups, then we examined the performance of deep GRNN model and other machine learning algorithms over each group of datasets. The overall results showed that the proposed model outperforms most machine learning models, especially over very high and high sparsity datasets.

$$SR = 1 - \frac{minority}{majority} \tag{2}$$

The rest of the present paper is structured as follows: Section two presents background about deep learning and data sparsity. Section 3 presents the related work. Section four presents the proposed model. Section five describes the employed datasets. Section six presents the used evaluation measures. Section seven shows the experimental setup and section eight shows the obtained results.

Section nine introduces threats to the validity of the study and finally, the paper ends with a section presenting the conclusion.

II. BACKGROUND

A. DATA SPARSITY

Variables with sparse data are those that have mostly zero values, which is different from variables with missing data [18]. Examples of sparse variables include vectors of one-hot-encoded words or counts of categorical data. On the other hand, variables with dense data have predominantly non-zero values. The problem of software defect density prediction is not considered as a complete data sparse problem because the input matrix is frequently dense, but the output variable is sparse. Sparse matrices can cause problems with regards to space and time complexity. It is time consuming to perform operations against input matrices with a high ratio of zero values. This is because such matrices inherently induce arithmetic computations involving multiplying or adding these zero values; the time complexity of this problem increases with larger input matrix sizes [18]. A reasonable solution to this problem is rather than handling the sparse output vector (i.e. defect density variable), we recommend to use deep neural networks to implicitly treat this issue.

B. AN OVERVIEW OF DEEP LEARNING

Deep learning is state of the art artificial intelligence algorithm, which consists of a family of algorithms including Recurrent Neural Network (RNN), Deep Neural Network (DNN), Convolutional Neural Networks (CNN), and Long-Short Term Memory (LSTM) [16], [17]. Deep learning has been intensively used in a variety of domains such as, natural language processing [19] and image processing. Deep learning is becoming increasingly prevalent in the field of software engineering [17], [20]. In this paper we focus on building a deep neural network model for predicting defect density. The DNN is an improved version of traditional artificial neural network with multiple dense layers. The DNN models are recently becoming very popular due to their excellent performance to learn not only the nonlinear input–output mapping but also the underlying structure of the input data vectors [20].

The DNN training process contains two learning passes (forward and backward passes) based on the backpropagation algorithm. In the forward pass, the input data is transformed to a specific form of output through layer by layer using nonlinear activation functions [21]. In the backward pass, the derivatives of the error function with respect to individual weights are updated in a reverse order, that is, from the output layer to the input layer. The Stochastic Gradient Descent is extensively employed throughout the training procedure for the sake of weights optimization. The DNN, on the other hand, necessitates tuning of various hyperparameters such as the number of neurons, hidden layers, and iterations, which might make solving a complex model computationally costly [11], [20], [21].

III. RELATED WORK

A. DEFECT DENSITY

Defect density is an important metric for measuring the efficacy of software development process. Various approaches have been used in recent years to estimate the defect density in software projects. Multiple studies employed statistical approaches to quantify and evaluate the link between defect density and available metrics. Correlation coefficients and determination, linear regression models, and multiple linear regression models are examples of statistical approaches used to assess the strength of that link.

The first research direction focuses on analyzing the relationship between defects and multiple of software metrics. In this regard, Rahmani and Khazanchi [9] developed four assumptions to examine the relationship between defect density and three source code metrics (number of downloads, number of developers, and size of the software). These metrics have been mined from 44 software projects. The statistical analysis on these assumptions showed that two out four assumptions were only accepted. The relationship between design metrics and defect density was also investigated by Mandhan et al. [22]. They applied seven design and code metrics and found that predicting defect density using the seven design and code metrics is statistically significant. On the other hand, Nagappan and Ball [23] used a group of code churn measures to predict defect density at early stages based on statistical regression models. The code churn is a measure or an indication of the change over time in the bulk of code within a software component. The results of Person and Spearman correlations confirmed that code churn measures are adequate for defect density prediction. They applied variety of machine learning methods to build defect density prediction model based on code churn. Verma and Kumar [24] extracted five metrics from multiple open source software projects. They proposed six hypotheses to examine the relationship between the defect density and the metrics. The statistical significance test revealed that four of the proposed hypotheses have been accepted. Sherriff et al. [25] built a prediction model for defect density prediction using five metrics. The constructed model was trained on 14 projects and tested on 6 projects. The results confirmed the relevancy of those five metrics for defect density predictions. They also introduced a method for predicting defect density in the source code for seven releases of the Glasgow Haskell Compiler. They used three types of metrics: 1) test metrics, 2) structural metrics, and 3) compiler warnings [26]. Marchenko [27] analyzed the relationship between code metrics and defect density in the field of embedded software development. They used CodeScanner and PC Lint to extract code metrics. They concluded that static code analysis tools can help agile teams to perform testing activities in a better way.

The second approach focuses on developing defect density prediction models from existing defect datasets. In this regard, Kutlubay et al. [28] used Naïve Bayes, and Decision Tree to predict the defect density from NASA projects. They demonstrated that the Decision Tree surpasses the regression models over nine datasets. On the other hand, Kumar et al. [13] and Khalsa [29] applied Fuzzy logic and neural networks based on three metrics extracted from 4000 log files. These metrics are: (1) defects in file pre-releases aggregated for a module, (2) total number of lines of code aggregated for a module, and (3) McCabe cyclamate complexity of a file aggregated for a module. They found that the results of neural networks are much better than results of the fuzzy inference model. Knab et al. [30] applied Decision Tree model over data collected from seven releases of the Mozilla open source.

They found that the number of modification reports is useful for defect density prediction, and the number of functions and lines of code have little predictive impact on defect density. López Martín et al. [10] applied support vector regression to predict defect density of software projects, where two support vector regression approaches were used based on 21 new projects. The accuracy of the two regression models was marginally good. They also developed advanced technique to predict defect density called Transformed K-nearest neighborhood output Distance Minimization (TKDM). The accuracy of the constructed model was compared to the previous support regression models and other machine learning models. The reported statistical results confirmed that the proposed model outperform other models. Yadav and Yadav [31] used fuzzy inference system to predict defect density using nine metrics. The predictive accuracy of the proposed model was validated using twenty projects. They found that the predictions generated by the fuzzy inference system are very close to the actual values.

B. DEEP LEARNING IN DEFECT PREDICTION

In the field of defect prediction, we found multiple studies that use deep learning (especially convolutional neural network) to predict bugs from log files. Most of these studies were designed to learn just-in-time defect predictions from log files such as DeepJIT model [32] and DeepCPDP [33]. Hoang al. [32] proposed a deep learning model to automatically discover embedding features from commit messages and code changes and use them to identify defects. Chen et al. [33] proposed a deep learning to predict defect prediction for cross projects. They represented the source code of each program module via a simplified abstract syntax. Qiao et al. [11] proposed a deep learning model to predict number of defects in the software module. Yang et al. [34] proposed a deep learning model for just -in-time defect prediction, the model was evaluated over six open source projects such Bugzilla, Platform, Columba, PostgreSQL and Mozilla.

Zhao et al. [35] proposed a deep learning technique called DeepSim to measure the functional similarities of code. They created code semantic representation from the encoded control flow and data flow to allow a deep neural network classification model to learn the embedded features from the representation matrix. Ma et al. [36] proposed an automated debugging technique based on deep learning called MODE. Their model mimics conventional debugging and regression testing and can perform model-state analysis to detect defects.

Guo et al. [37] applied RNN and word embedding to produce code trace links. The extracted word embedding vectors are used by the RNN to learn the sentence semantics. On the other hand, Gu et al. [38] developed a deep learning model to perform code searches. A high-dimensional vector space is used to represent the code snippets and natural language descriptions. Furthermore, they were the first to use RNN to generate API sequence suggestions.

White et al. [39] proposed a code suggestion model based on feed forward network and RNN. They reported that the deep learning can create high-quality models from a corpus of Java projects. In a different study, they proposed another model to detect code clone. The constructed mode has ability to automatically find discriminating features in the source code. They reported that all the parts of the source code can be represented and used for clone detection.

Huo et al. [40] used a CNN to discover the most likely defected code based on a bug report based on semantic features. They applied both lexical and program structural information to discover the semantic features from source code and natural language to make bug localization. Jana et al. [41] used a deep neural network to generate test cases automatically that will be used for automated testing of erroneous behaviors of DNN-controlled vehicles. Lam et al. [42] developed a new combination approach between revised vector space model and deep neural networks to perform bug localization. The DNN is used mainly to learn the terms in the bug reports and source files.

Above all, we can find that no prior studies used module defect density. All of them used complete defect density of the project. Therefore, those studies did not consider the problem of data sparsity in their approaches because it was not exist. Furthermore, no studies used deep learning in predicting defect density where the existing deep learning studies were designed for the problem of defect prediction that aims to classify the software modules as defective or not. Machine learning and statistical learning methods were the dominant for predicting the defect density. These reasons form the main motivation for carrying out this research study.

IV. THE PROPOSED DEEP NEURAL NETWORK MODEL

This section describes the proposed deep neural networks for predicting defect density at module level. The proposed model is an enhanced form of a popular type of artificial neural network that is called General Regression Neural Network (GRNN). The GRNN would be formed instantly with just a 1-pass training with the training data. It is a four layers network used mainly for regression tasks as shown in Fig. 1. This kind of network consists of input and output layers which are responsible for receiving input and producing regression output, in addition to two hidden layers. The first hidden layer

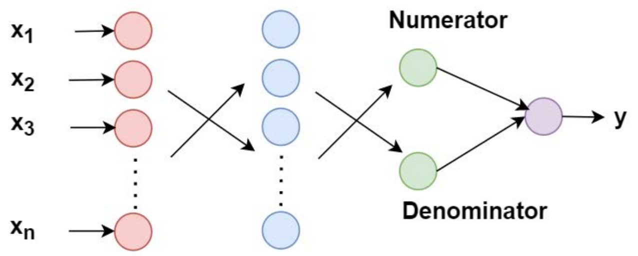

FIGURE 1. General regression neural network.

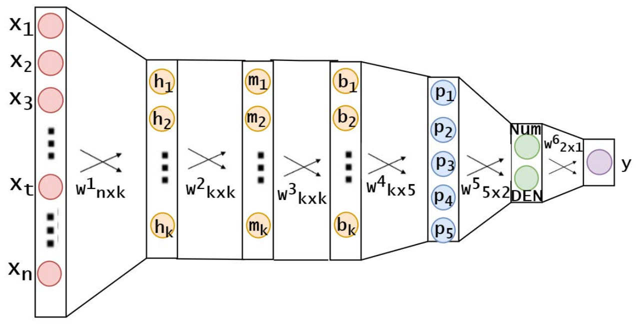

FIGURE 2. The enhanced deep GRNN (DGRNN).

is known as pattern layer and responsible for pattern discovery, and the second hidden layer is known as summation layer which consists of two neurons (numerator and denominator summation). The denominator summation calculates the sum of the weights coming from each neuron in the second layer. The numerator summation calculates the sum of the weights multiplied by the actual output of each pattern neurons. The activation function in the neurons of the of the first hidden layer is usually RBF function that use Euclidean distance between input vector and the neuron's center to discover the hidden pattern in the training data. However, the GRNN does not need iterative training algorithm, but in contrast it approximates any arbitrary function between input and output data.

In order to enhance GRNN to become Deep GRNN (DGRNN) model, we added extra three dense hidden layers before pattern hidden layer in order to extract hidden features in the training data as shown in Fig. 2. The newly designed deep neural network (i.e., DGRNN) composed of seven layers: input layer, 5 hidden layers and output layer. The first and third hidden layers (denoted as h and b in Figure 2) are built using 10 neurons each, with Radial Basis activation function (RBF). Whilst, the second hidden layer (denoted as m) is a drop out layer to prevent over fitting. The pattern layer is constructed using 5 neurons to discover the complex patterns in the data, in this layer we use Radial Basis Function as activation function. The summation hidden layer is composed of two neurons as in the original GRNN in order to facilitate producing the final regression output. (i.e., defect density). As shown in the figure, there are multiple weight matrices (5 matrices between layers) that will be updated during training process. The complete network will be trained using gradient descent algorithms.

TABLE 1. Summary of 28 datasets description.

Dataset	# Instances		# metrics	Average of defect density per 1K LOC	MAE of random guessing	SD of random guessing
	%sparsity	All				
jedit-3.2	66.90%	272	20	1.40	11.9	81.7
jedit-4.2	86.90%	367	20	0.29	1.3	6.8
log4j-1.1	66.06%	109	20	0.79	30.1	13.8
JDT_R2_0	54.03%	2397	48	1.57	72.4	87.5
JDT_R2_1	68.06%	2743	48	0.78	5.7	9.0
JDT_R3_0	61.40%	3420	48	1.31	8.5	14.6
JDT_R3_1	67.24%	3883	48	1.08	7.1	11.9
JDT_R3_2	63.46%	2233	48	1.04	7.9	12.7
PDE_R2_0	80.73%	576	48	0.42	3.9	9.7
PDE_R2_1	83.71%	761	48	0.30	3.6	8.7
PDE_R3_0	68.79%	881	48	0.66	7.0	10.6
PDE_R3_1	67.78%	1108	48	0.70	6.9	10.7
PDE_R3_2	53.81%	1351	48	0.83	10.7	12.8
arc	88.46%	234	20	0.13	8.6	35.8
ant-1.7	84.00%	125	20	0.45	2.9	10.4
redaktor	84.66%	176	20	0.15	1.5	3.2
xalan-2.4	84.79%	723	20	0.22	2.8	19.7
xerces-1.2	83.86%	440	20	0.26	74.4	27.0
camel-1.0	96.16%	339	20	0.04	7.2	74.7
camel-1.6	80.52%	965	20	0.52	31.1	14.3
ivy-2.0	93.36%	241	20	0.11	3.2	21.3
prop-6	90.00%	660	20	0.09	4.3	27.7
poi-2.0	88.22%	314	20	0.12	2.2	11.6
lucene-2.0	53.34%	195	20	1.37	50.1	29.5
synapse-1.0	72.98%	222	20	0.13	3.1	21.5
synapse-1.2	66.41%	256	20	0.57	10.4	45.1
velocity-1.6	65.94%	229	20	0.83	23.6	61.7
xerces-1.3	84.77%	453	20	0.43	22.1	15.5

V. DATASET

Fortunately, there are dozens of benchmark datasets available in different software repositories which facilitates evaluating prediction models and generalize our conclusions. However, since we are interested only in defect density, the datasets that do not have bug counts variables are excluded because we cannot calculate defect density in this case. This has resulted in 28 public datasets collected from three main sources: AEEEM repository [43], [44], SOFTLAB [14] and MORPH [45]. Most of the datasets share the same feature descriptions. Mainly, they use McCabe metrics, Halsted metrics to describe software modules (see the Appendix for more details about these metrics). But, since these datasets do not have defect density, we derived this variable by applying equation 1 on each dataset. The complete list of the employed datasets and their characteristics are described in Table 1, in which we can observe that most datasets have defected ratio less than 30% which means that the target defect density variable will often be sparse due to the size of non-defective modules. This is also confirmed by histogram of defected ratio in Fig. 3. Therefore, as part of our investigation is to examine the role of sparsity on the performance of defect density prediction models.

To get insight into the relationship between the average of defect density and the number of instances in the datasets, we draw a scatter plot between two variables as shown in Fig. 4. We can observe that small datasets with less than

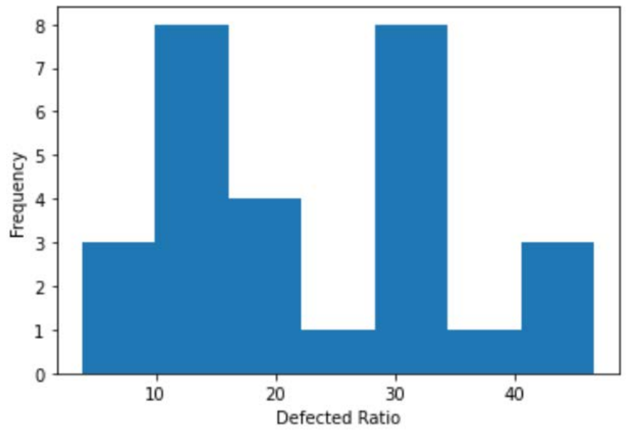

FIGURE 3. Histogram of defected ratio.

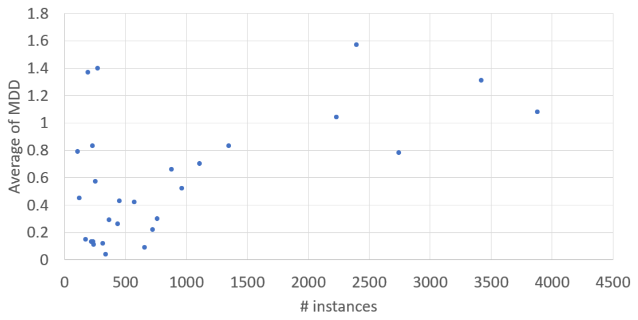

FIGURE 4. Relationship between the average of DD and number of instances in the dataset.

1000 instances have frequently a small average of defect density (i.e., less than 1 defect/KLOC), while large modules have more than 1 defect/KLOC. This confirm that the relationship between size of the module and its defect density is generally positive.

As a part of our investigation is to analyze the role of SR (see equation (1)) with deep learning, the 28 datasets are divided into four groups based on inter-quartile range of the SR as shown in Table 2. The labels are chosen to align with the broader context of data sparsity. Interestingly the number of datasets in each bin is different and most of them belong to medium group. Fig. 5 shows the SR histogram for all datasets. As we can note, the majority of the data sets were gathered around the median in the SR range 0.49 to 0.84. The sparsity is usually categorized in terms of SR orders of magnitude. We find that only 14 out of 28 data sets have SR greater than the median (SR=0.73). In other words, the data sparsity level of 73.0% software defect density is not as extreme as in some other domains. This is also an indication of our study that very high and high levels of SR are most common.

VI. EVALUATION MEASURES

Evaluation measures are used to measure the success of prediction models. In the literature, there are multiple evaluation measures, we only limited our choice to three popular measures that exhibit unbiased nature and present systematic error distribution. These measures are Normalized Mean Absolute Error (MAE), Standardized Accuracy (SA), and Effect Size

TABLE 2. Datasets groups based on sparsity ratio.

SR Range	Bin Name	# datasets	Datasets
(0.84, 0.96]	Very High (VH)	7	jedit-4.2, arc, camel-1.0, ivy-2.0, prop-6, poi-2.0, synapse-1.0
(0.73, 0.84]	High (H)	7	PDE_R2_0, PDE_R2_1, redaktor, xalan-2.4, xerces-1.2, camel-1.6., xerces-1.3
(0.49, 0.73]	Medium (M)	9	jedit-3.2, log4j-1.1, JDT_R2_1, JDT_R3_1, PDE_R3_0, PDE_R3_1,ant-1.7, synapse-1.2, velocity-1.
(0.13, 0.49]	Low (L)	5	JDT_R2_0, JDT_R3_0, JDT_R3_2, PDE_R3_2, lucene-2.0

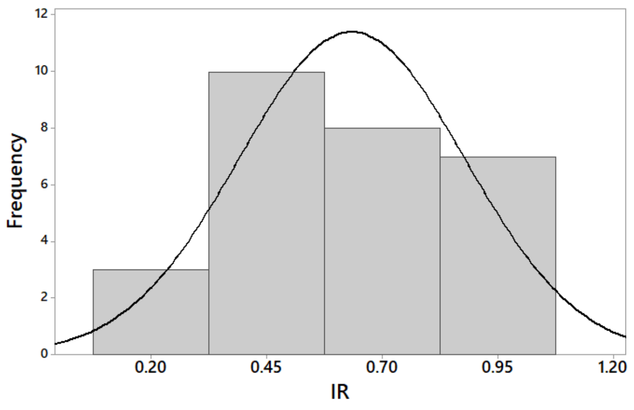

FIGURE 5. Distribution of data Sparsity Ratio (SR).

(ES) [46], [47]. The MAE is calculated by computing the average of the absolute errors dived by the summation of actual defect density to facilitate comparison across different datasets as shown in equation (3). The SA measure is used to investigate whether the prediction model can surpass a baseline model as shown in equation (4). Usually, the baseline model is given as random guesses, therefore the results obtained by SA can also be interpreted as a measure of improvement. We also support SA with the effect size (ES) to see whether these improvements are made by chance as shown equation (5). To interpret the result of ES, one can use the default scale given by Shepperd et al. [46] which report that if ES is around 0.2 then the effect is small, medium if ES around 0.5 and large of ES about or greater than 0.8. The MAE and SD of random guessing for each dataset is reported in the last two columns of Table 1. These values will be used to evaluate the performance of each model over each dataset.

$$MAE = \frac{\sum_i^n |DD_i - \widehat{DD_i}|}{\sum_i^n DD_i} \tag{3}$$

$$SA = 1 - \frac{MAE}{MAE_r} \tag{4}$$

$$ES = \frac{MAE - MAE_r}{SD_r} \tag{5}$$

where: DD_i is the actual defect density of the i^{th} module, and $\widehat{DD}_i$ is the predicted defect density. MAE_r and SD_r are the mean absolute error and standard deviation of the random generated errors at $r = 1000$ run.

VII. EXPERIMENT SETUP

A. DATA PRE-PROCESSING

Pre-processing the data is an essential step before building the prediction model to ensure its reliability and avoid possible bias in the predictions. Mainly, we transformed all numerical variables by applying min-max scaling to every variable of the data to bring each feature's values in the interval [0, 1]. All categorical variables are transformed to numerical variables by applying one-hot encoder method. Concerning outliers, we used box plots to determine the observation with extreme values. In this case we remove the observations that have extreme value in at least two variables. Finally, all observations with missing values are imputed using average imputation for numerical variables and frequent value imputation for the categorical variables.

B. CHOICE OF MACHINE LEARNING MODELS

The proposed DGRNN model has been compared against a set of popular baseline machine learning regression methods. These methods are Support vector regression, k-nearest neighbor, Multi-layer perceptron, Extreme Gradient Boost (XGB) and AdaBoost, Random Forest, Multi-linear regression, and GRNN. Finally, we used the default configuration parameter for each machine learning algorithm.

C. VALIDATION PROCEDURE

A 10×10 folds cross-validation was used to evaluate and compare between the different prediction models. The main reason for this choice is to reduce the bias in selection of training and test data [48]. According to Kocaguneli et al. [49], unlike leave one cross validation using 10-Folds cross-validation generates estimates of test error with higher bias, and lower variance. Therefore, using the repeated 10-Folds cross validation would generally reduce such bias and keep variance very small. However, the prediction model is developed on the training data while the testing data is used to evaluate the model. The error measures are calculated for each test dataset then aggregated overall sets. This procedure continues until all examples within the dataset run as test.

To better train and tune the DGRNN model for each dataset, we adjust the different configuration parameters such as the optimizer, number of epochs and the dropout value. The best choice for these parameters is a difficult process. thus, we tried specific values for each parameters and the values that improve the accuracy of the model are then chosen. Here are the parameter values we used:

- **Optimizer**: different optimizers (SGD, ADAM and ADAMW) have been tried as a part of the training process.Each combination of dataset and model can select different optimizer.

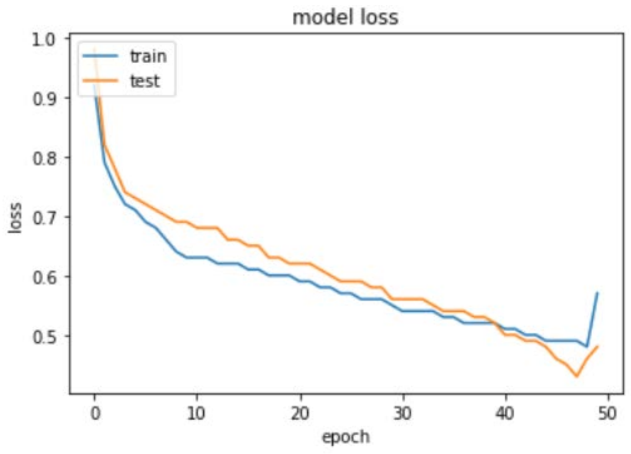

FIGURE 6. Training and validation loss curves of DGRNN on JDT_R2_1 dataset.

- **Number of epochs**: many numbers of epochs from 1 to 50 have been applied. In addition, the stop accuracy method that allowed us to stop the training when we reached the highest accuracy has been applied, so as not to waste time and storage.
- **Spread or Sigma**: The spread is hypermeter of GRNN, it is required careful selection. we have tried three recommended values, 0.1, 0.2 and 0.3
- **Dropout value**: we tried 0.1, 0.25 and 0.5, with each value producing a slightly different result.

During the training procedure, we divided our training data randomly into 70% training and 30% validation for the sake of finding out the best value of spread (sigma). The sigma value is used within RBF to governing the smoothness of a GRNN. The best practice is to find the value of sigma when the MAE is minimum. Figure 6 shows the training and validation loss curves of DGRNN on JDT_R2_1 dataset. We could not add all figures because simply we have around 100 loss figure for each dataset. This figure can give us insight on how DGRNN model is trained and validated. We can see that there is overfitting after epoch 40, this was very common for all training models therefore we reduce number of epochs to 40 instead of 50. However, the training and validation time of DGRNN was considered adequate and minimal for all dataset in comparison to conventional GRNN and other machine learning algorithms. The reason for that is the spread (Sigma) coefficient is the only parameter that need to identified by the train and validation procedure.

VIII. RESULTS AND DISCUSSION

This section comments of the findings and results obtained after conducting empirical validation for the sake of addressing the research questions.

RQ1: *Does Deep learning outperform the traditional machine learning algorithms for defect density?*

To address this question, we compared our constructed DGRNN to the benchmark algorithms mentioned in the subsection VII-B. The evaluation results are presented in terms of the normalized MAE in addition to the standardized accuracy

TABLE 3. Summary of overall accuracy results for within and cross projects datasets.

	MAE	SA	ES
ADA	0.67 (0.52 - 0.82)	0.64	0.24
DGRNN	0.41 (0.28 - 0.58)	0.78	0.39
GRNN	1.01 (0.77 - 1.48)	0.61	0.27
KNN	0.95 (0.81 - 1.10)	0.47	0.15
MLP	1.29 (1.13 - 1.44)	0.3	0.13
MLR	2.01 (1.65 - 2.38)	-0.09	0.04
RF	0.57 (0.38 - 0.76)	0.7	0.27
SVM	0.90 (0.86 - 0.95)	0.5	0.16
XGB	0.56 (0.37 - 0.76)	0.7	0.27
ZIFR	0.92 (0.83 - 1.00)	0.5	0.17

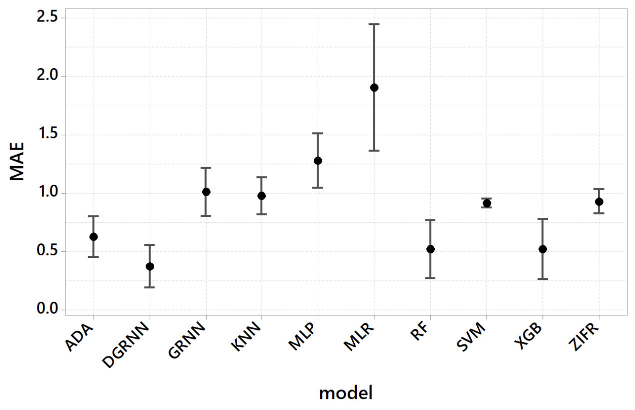

FIGURE 7. Interval Plot of MAE at 95% CI for within project.

and effect size (ES). However, in addition to the point estimate of MAE, we reported the confidence interval of MAE distribution as shown in Table 3.

The results in Table 3 are analyzed from two aspects: accuracy and reliability. The three evaluation measures are used mainly to differentiate between models, whereas the SA and ES are used to examine the reliability of the proposed model against random guessing. These results are obtained after running each model over 28 datasets. Therefore, these results represent the average of each accuracy measure across all datasets. From Table 3, we can see that all models, except MLR, produce predictions better than random guessing as confirmed by SA and ES. We can observe that six out of nine models obtained SA greater or equal 0.5, which means they produce meaningful predictions, better than random guessing. However, the effect size show that most of the improvements have small effect with ES about 0.25 or less. A large value of SA (i.e., 0.78) means that our model can produce reliable predictions that are better than random guessing and confirm that there is a significant improvement. On the other hand, the effect size of DGRNN (ES = 0.39) indicates that the obtained improvement is rather significant (not by chance). It is interesting to note that the DGRNN produced better accuracy than conventional GRNN. Other accuracy results revealed that RF and XGB are competitors for DGRNN, MLR and MLP were two of the worst models with very poor average MAE values. These finding raises concerns about the adequacy of these model for such complex structure datasets. Also, it shows that traditional machine learning usually suffer from sparse output variable and yield bias predictions.

The accuracy results in terms of the normalized MAE demonstrate that our proposed DGRNN model has capability to produce much better accuracy than other employed machine learning models, especially conventional GRNN. Since we only report the average of MAE in Table 3, we need to understand the distribution of MAE across all used datasets. Therefore, we plotted the interval plot for MAEs at 95% Confidence Interval (CI) as shown in Fig. 7. The interval plot has been adopted for two reasons: 1) It is a graphical data interpretation tool that allows us to visually examine the differences between several groups and make preliminary conclusions. 2) Because we're looking for substantial differences between the means of different MAE populations, the point mean estimate in Table 3 is not enough to determine whether the multiple MAE distributions are significantly different. This means that for each sample, the average, standard deviation, and sample size are combined to create a confidence interval that represents a given level of confidence, such as 95%. Under these circumstances, it is predicted that the population mean will be included in 95% of the sample's confidence intervals. From Fig. 7, we can simply observe that the MAE distribution of DGRNN is significantly better than all eight prediction models, except three (ADA, RF and XGB). Remarkably, the MLR model is not useful for the defect density predictions due to the large variability in MAE distribution. Both KNN and GRNN are relatively have same MAE distributions. We can also notice that the defect density prediction performance is negatively impacted when using SVM and ZIFR even though they have the smallest confidence intervals.

In addition to the above analysis, we reported the average improvement in DGRNN model against other defect density prediction models. Practically, we compute the SA and effect size for DGRNN considering that the comparative model as a baseline model. Referring to equations (4) and (5), we consider DGRNN as the model under evaluation, and the comparative model is the baseline model (MAE_r). Table 4 presents the results of the models' evaluation in addition to the Wilcoxon signed-rank significance test results obtained by comparing MAE distributions of DGRNN with each comparative model. From Table 4, we can observe that DGRNN model has a large significant improvement of 0.53 against GRNN, 0.55 against KNN, 0.79 against the MLR and as low as 0.25 against RF and XGB. These improvements are generally supported by a medium and very large effect size. These results show the significant improvements in defect density prediction when using the DGRNN model. The significance test also confirms that the DGRNN modules produced better predictions than most models except for RF and XGB. However, we can still see a good improvement of DGRNN over RF, XGB w.r.t SA and ES.

TABLE 4. Improvement of DGRNN over each one of the base models.

Baseline model	p-value	SA	ES
ADA	**0.02**	0.36	0.62
GRNN	**<0.001**	0.53	1.29
KNN	**<0.001**	0.55	1.44
MLP	**<0.001**	0.67	2.19
MLR	**<0.001**	0.79	1.67
RF	0.24	0.25	0.29
SVM	**<0.001**	0.53	4.23
XGB	0.25	0.25	0.27
ZIFR	**<0.001**	0.54	2.19

In summary, the answer to this question is yes. As reported in Table 4 and Fig. 7, the proposed DGRNN model significantly surpasses the seven machine learning models (KNN, GRNN, MLP, MLR, ADA, SVM, and ZIFR) based on the Wilcoxon signed-rank test. However, we have concerns about DGRNN's performance against RF and XGB which shows there is no significant difference. But it is still competitive with these ensemble learning models, and even better with average MAE as shown in Fig. 7.

RQ2: *What is the role of data sparsity in defect density prediction based deep learning?*

To answer this question, we study the role of data sparsity levels for the defect density variable in the performance of DGRNN and other prediction models. Since the used datasets contain few numbers of defected modules, the majority of records have zero-defect density. Therefore, we investigate the impact of this challenge on the performance of the proposed model. Conventionally, the modules with zero defect density are known as majority and the remaining modules are known as minority. The sparsity ratio is usually computed in equation (2), a higher value indicates a large sparsity level.

Figures 8 and 9 show the MAE values under various values of SR. The shadow region indicates the confidence range of 0.95 for the line created by a non-parameter smoother (loess smoothing). We can see a lot of diversity in predictive performance for any given amount of sparsity in this revised scatter plot, but we can also see that SR has a detrimental influence on predictive performance. The smoothed line begins at almost MAE = 1.2, drops to 0.9 at the median, and then increases to 1.7 MAE when the SR surpasses 80%. In other words, classifiers outperform random guessing at this level of SR (see effect size of VH category in Table 5). In general, the negative impact is seen on the right side of the dotted line for extremely high SR data sets, as well as on the left side of the dotted line for low SR data sets, where sparsity can cause a large decline in predictive performance. On the other hand, we can see the similar tendency if we simply look at DGRNN's accuracy with different levels of data sparsity, however the degree of unpredictability for lower SR levels is fairly considerable. This implies a negative relationship between performance and SR, which is considered a substantial influence in most cases. In conclusion, the very high levels of data sparsity in software defect data threatens the accuracy of prediction models. Therefore, any method of manipulating sparsity in the data is likely to be useful for predicting software defect.

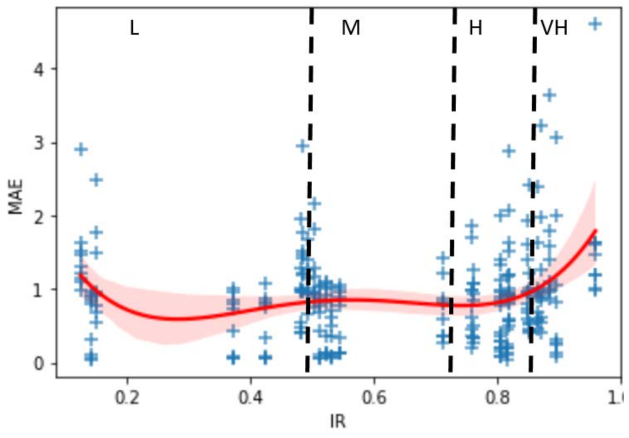

FIGURE 8. Performance of prediction models under differing levels of data sparsity.

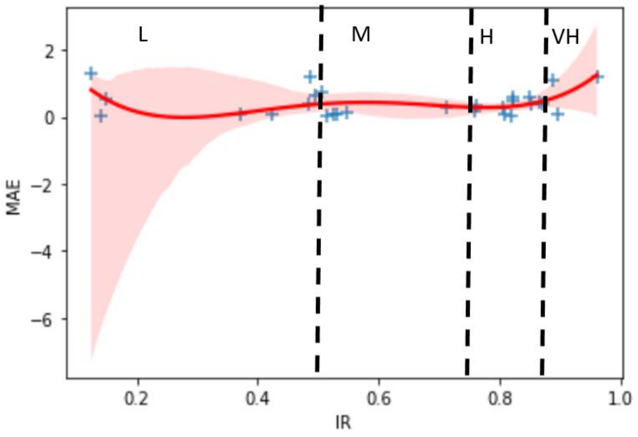

FIGURE 9. Performance of DGRNN model under differing levels of data sparsity.

Table 5 summarizes the accuracy results of the nine models under different levels of data sparsity. The values in boldface represent the most accurate results. We can observe that the proposed model beats the others under all datasets and other dataset categories. This provides evidence that the proposed DGRNN model can work perfectly under different levels of data sparsity with good accuracy. Also, the average improvement and the size of its effect for DGRNN on random guessing are remarkable and a good indication of the superiority of the proposed model.

We also examined the average improvements of DGRNN against other prediction models across different levels of data sparsity. In this evaluation, the DGRNN is considered the model under evaluation and each one of the comparative models is considered as baseline model. Interestingly, we noticed a significant degree of average improvement against, SVM, MLR, and MLP across all the different levels of SR. Similar patterns can be observed across all levels of SR datasets as our model can significantly surpass five machine learning

TABLE 5. Summary of accuracy for all prediction models across different categories of datasets.

Model	Very High			High			Medium			Low		
	MAE	SA	ES	MAE	SA	ES	MAE	SA	ES	MAE	SA	ES
ADA	0.8	0.59	0.1	0.52	0.73	0.23	0.65	0.63	0.3	0.73	0.59	0.33
DGRNN	**0.6**	0.59	0.1	**0.29**	0.78	0.24	**0.4**	0.72	0.34	**0.4**	0.72	0.4
GRNN	1.30	1.05	0.18	0.81	0.67	0.15	1.02	0.77	0.41	0.82	0.74	0.25
KNN	0.81	0.58	0.1	0.82	0.57	0.18	1.18	0.32	0.15	0.93	0.44	0.18
MLP	1.68	0.14	0.03	1.32	0.3	0.1	1.07	0.39	0.18	1.07	0.37	0.2
MLR	3.03	-0.56	-0.08	1.8	0.05	0.04	1.58	0.11	0.1	1.67	0.04	0.13
RF	0.83	0.58	0.1	0.41	0.78	0.25	0.54	0.71	0.35	0.47	0.74	0.41
SVM	0.94	0.52	0.09	0.89	0.53	0.16	0.89	0.49	0.21	0.91	0.45	0.2
XGB	0.82	0.58	0.1	0.34	0.82	0.26	0.55	0.7	0.34	0.54	0.7	0.4
ZIFR	1.06	0.46	0.08	0.8	0.58	0.18	0.84	0.52	0.22	1.01	0.41	0.22

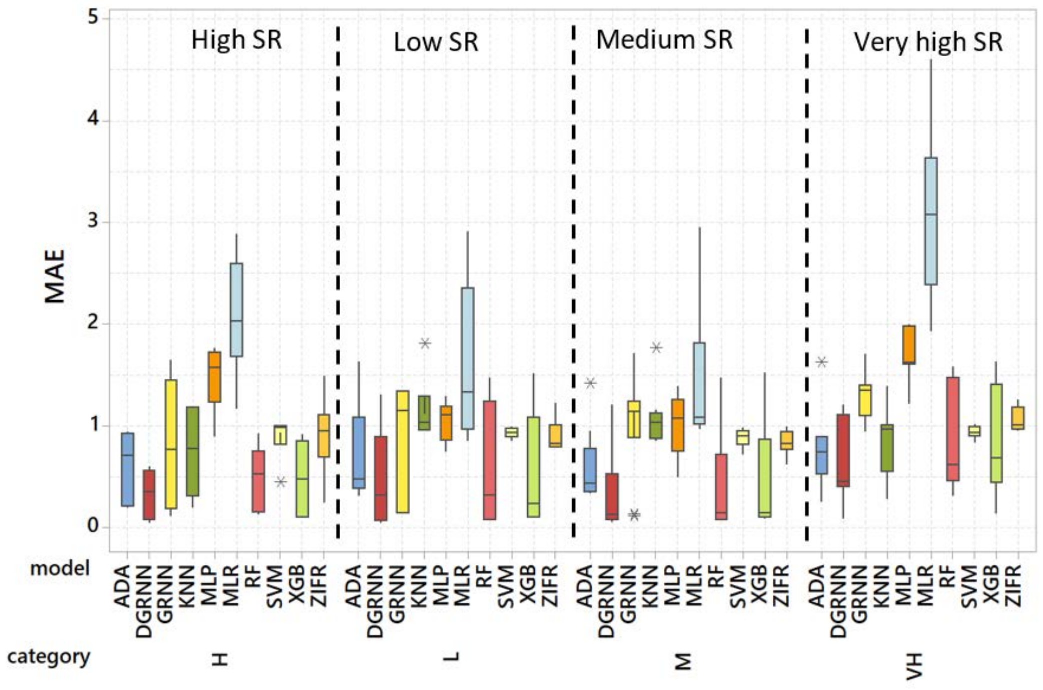

FIGURE 10. Boxplots of prediction models across each SR dataset category.

models and produce a relatively acceptable improvement over ensemble learning models such as RF, ADA, and XGB.

Figure 10 shows a comparative analysis of all defect density prediction models with boxplot visualizations. SR values are used as distinguishing factors between various dataset groups (from Low to Very High); these are depicted using dashed vertical lines. Although, the consistency between boxplots indicates little differences across SR levels, nonetheless our proposed model fairs well in comparison to other models especially against XGB and RF models. The smaller box heights in the SVM and ZIFR models indicate smaller variability in these models, the prediction accuracy within these two models is underestimated. For datasets with Very High SR levels, we can see that KNN and ADA models are much more competitive and very close to the top three models (XGB, RF and DGRNN); in this scenario, our model produced a smaller MAE median; in fact, when it comes to the median of MAE metric, our model is consistently better than the others and KNN is the least stable model dues to very high MAE variability and fluctuation.

In summary, we conclude that the DGRNN model is stable and less sensitive to changes in SR levels as seen in Figure 10 and Table 6. However, the DGRNN and ensemble learning models are comparative but not significant even though the DGRNN model produces good average improvements in SA and the effect size on these models.

IX. THREATS TO VALIDITY

In this section we describe the main threats to our study validity. There are two main types of threats to validity: 1) Internal validity and 2) External validity. Internal threats to validity mainly deal with validation approach and choices of prediction models. It is well known that the use of validation technique techniques to empirically validate the prediction

TABLE 6. Average improvement of DGRNN over each one of the base models.

Baseline Model	Very High			High			Medium			Low		
	p-value	SA	ES	p-value	SA	ES	p-value	SA	ES	p-value	SA	ES
ADA	0.4	0.24	0.45	0.13	0.43	0.81	0.15	0.39	0.69	0.43	0.45	0.58
GRNN	0.09	0.23	0.47	**0.02**	0.71	1.1	**<0.001**	0.61	1.8	**0.04**	0.52	2.1
KNN	0.33	0.26	0.56	**0.04**	0.64	1.3	**<0.001**	0.66	2.18	**0.02**	0.57	4.23
MLP	**<0.001**	0.64	4.01	**<0.001**	0.78	3.37	**<0.001**	0.63	2.21	**0.03**	0.62	1.61
MLR	**<0.001**	0.8	2.69	**<0.001**	0.84	2.52	**<0.001**	0.75	1.74	**0.017**	0.76	1.32
RF	0.38	0.27	0.44	0.34	0.28	0.42	0.674	0.25	0.25	0.75	0.15	0.12
SVM	**0.047**	0.35	5.43	**0.002**	0.67	3.02	**0.002**	0.55	5.46	**0.047**	0.56	7.2
XGB	0.41	0.26	0.4	0.41	0.15	0.19	0.589	0.28	0.28	0.87	0.25	0.21
ZIFR	**0.02**	0.43	3.93	**0.015**	0.63	1.96	**0.004**	0.53	3.73	**0.049**	0.6	1.84

model has significant impact of the final results. Although there is a debate about each type of validation techniques, we favor using repeated 10 Folds cross validation. This approach can reduce potential bias from data sampling. The second threat is the choice of machine learning algorithms that will be compared against our proposed model. We chose a set of popular machine learning prediction methods in order to compare them with DGRNN.

On the other hand, the external validity deals with two main threats: choice of datasets and evaluations measures. In terms of datasets, we decided to use large portion of public datasets, but unfortunately not all defect datasets can fit for regression problem as in defect density problem due to the absence of bug count variable. Therefore, these datasets were excluded from our empirical investigation, even though they might affect our derived conclusions. With respect to the evaluation measures, Shepperd and McDonnell [46] argued that the relative accuracy measures usually yield bias conclusion and affect the generalizability of the findings. Therefore, we decided to use only the most trustworthy measures such as MAE, SA and effect size because they are unbiased and can tell us the true conclusion when comparing the prediction models [2], [50], [51].

X. CONCLUSION

Delivering a high-quality software product is an essential task during software testing and maintenance phases. Defect density is an important factor of software product quality. In this paper we proposed an enhanced deep neural network called DGRNN model based on the traditional GRNN neural network to predict defect density. The constructed DGRNN model has been evaluated against other popular machine learning models using the repeated 10-Folds cross validation, and 28 public datasets from different repositories. Since defect density variable in these datasets are usually sparse, we examined the role of data sparse levels on the performance of DGRNN and other prediction models. The obtained results demonstrate that the DGRNN model significantly surpasses the other prediction models over 28 datasets, and especially over datasets with high and very high SR levels. Furthermore, our model is competitive to other ensemble learning such as RF, XGB and Ada over medium and low SR levels. However, we also found that the use of MLP and MLR models harm the accuracy of defect density prediction. Finally, we concluded that the performance of the prediction models is severely threatened by very high levels of data sparsity in the software defect dataset. Therefore, any method of manipulating sparsity in the data is likely to be useful for predicting defect density.

APPENDIXES

DESCRIPTION OF SOFTWARE METRICS

1) **Halsted and McCabi static code metrics**
   - OP: The number of operators
   - OD: The number of operands
   - UOP: The number of unique operators
   - UOD: The number of unique operands
   - C_LTH: Halstead Length is the total of all the lengths in the methods
   - C_VOL: Halstead Volume is the total of all the volumes in the methods
   - C_BUG: Halstead Bugs = C_VOL/3000
   - C_EFF: Halstead Effort = is the total of all the effort in the methods
   - v(G): Cyclomatic Complexity Metric
   - ac: Actual Complexity Metric
   - iv(G): Module Design Complexity Metric
   - ev(G): Essential Complexity Metric
   - pv(G): Pathological Complexity Metric
   - S0: Design Complexity Metric
   - S1: Integration Complexity Metric
   - OS1: Object Integration Complexity Metric
   - gdv(G): Global Data Complexity Metric
   - DV: Data Complexity Metric
   - DR: Data Reference Metric
   - TDR: Tested Data Reference Metric
   - maint_severity: Maintenance Severity Metric
   - DR_severity: Data Reference Severity Metric
   - DV_severity: Data Complexity Severity Metric
   - gdv_severity: Global Data Severity Metric
   - MAXV: Maximum v(G)

- MAXEV: Maximum ev(G)
- QUAL: Hierarchy Quality

2) **CK metrics**

- WMC: weighted methods per class
- DIT: depth of Inheritance Tree
- NOC: number of children
- CBO: coupling between object classes
- RFC: response for a class
- LCOM: lack of cohesion in methods
- LCOM3: lack of cohesion in methods, different from LCOM
- NPM: number of public methods
- DAM: data access metric
- MOA: weighted methods per class
- MFA: measure of functional abstraction
- CAM: cohesion among methods of class
- IC: inheritance coupling
- CBM: coupling between methods
- AMC: average method complexity
- AC: afferent couplings
- EC: efferent couplings
- Max(CC): max value of McCabe's cyclomatic complexity
- Avg(CC): average value of McCabe's cyclomatic complexity
- LOC: lines of code

REFERENCES

[1] T. Menzies, A. Butcher, D. Cok, A. Marcus, L. Layman, F. Shull, B. Turhan, and T. Zimmermann, "Local versus global lessons for defect prediction and effort estimation," *IEEE Trans. Softw. Eng.*, vol. 39, no. 6, pp. 822–834, Jun. 2013.

[2] C. Tantithamthavorn, A. E. Hassan, and K. Matsumoto, "The impact of class rebalancing techniques on the performance and interpretation of defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 46, no. 11, pp. 1200–1219, Nov. 2020.

[3] D. Bowes, T. Hall, and J. Petrić, "Software defect prediction: Do different classifiers find the same defects?" *Softw. Quality J.*, vol. 26, no. 2, pp. 525–552, 2018.

[4] D. Tomar and S. Agarwal, "Prediction of defective software modules using class imbalance learning," *Appl. Comput. Intell. Soft Comput.*, vol. 2016, pp. 1–12, Jan. 2016.

[5] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Trans. Softw. Eng.*, vol. 34, no. 4, pp. 485–496, Jul. 2008.

[6] W. Fei, X.-Y. Jing, S. Ying, S. Jing, and Y. Sun, "Cross-project and within-project semisupervised software defect prediction: A unified approach," *IEEE Trans. Rel.*, vol. 67, no. 2, pp. 581–597, Jun. 2018.

[7] C. López-Martín, Y. Villuendas-Rey, M. Azzeh, A. Bou Nassif, and S. Banitaan, "Transformed k-nearest neighborhood output distance minimization for predicting the defect density of software projects," *J. Syst. Softw.*, vol. 167, Sep. 2020, Art. no. 110592.

[8] P. Mohagheghi, R. Conradi, O. M. Killi, and H. Schwarz, "An empirical study of software reuse vs. defect-density and stability," in *Proc. 26th Int. Conf. Softw. Eng.*, 2004, pp. 282–291.

[9] C. Rahmani and D. Khazanchi, "A study on defect density of open source software," in *Proc. IEEE/ACIS 9th Int. Conf. Comput. Inf. Sci.*, Aug. 2010, pp. 679–683.

[10] C. Lopez-Martin, M. Azzeh, A. Bou-Nassif, and S. Banitaan, "Upsilon-SVR polynomial kernel for predicting the defect density in new software projects," in *Proc. 17th IEEE Int. Conf. Mach. Learn. Appl. (ICMLA)*, Dec. 2018, pp. 1377–1382.

[11] L. Qiao, X. Li, Q. Umer, and P. Guo, "Deep learning based software defect prediction," *Neurocomputing*, vol. 385, pp. 100–110, Apr. 2020.

[12] S. Rathaur, N. Kamath, and U. Ghanekar, "Software defect density prediction based on multiple linear regression," in *Proc. 2nd Int. Conf. Inventive Res. Comput. Appl. (ICIRCA)*, Jul. 2020, pp. 434–439.

[13] V. Kumar, A. Sharma, and R. Kumar, "Applying soft computing approaches to predict defect density in software product releases: An empirical study," *Comput. Inform.*, vol. 32, no. 1, pp. 203–224, Mar. 2013.

[14] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano, "On the relative value of cross-company and within-company data for defect prediction," *Empirical Softw. Eng.*, vol. 14, no. 5, pp. 540–578, 2009.

[15] L. L. Minku, "A novel online supervised hyperparameter tuning procedure applied to cross-company software effort estimation," *Empirical Softw. Eng.*, vol. 24, no. 5, pp. 3153–3204, Feb. 2019.

[16] I. H. Sarker, "Deep learning: A comprehensive overview on techniques, taxonomy, applications and research directions," *Social Netw. Comput. Sci.*, vol. 2, no. 6, p. 420, Nov. 2021.

[17] S. Wang, T. Liu, J. Nam, and L. Tan, "Deep semantic feature learning for software defect prediction," *IEEE Trans. Softw. Eng.*, vol. 46, no. 12, pp. 1267–1293, Dec. 2020.

[18] Y. Zhu, X. Wu, J. Qiang, X. Hu, Y. Zhang, and P. Li, "Representation learning with deep sparse auto-encoder for multi-task learning," *Pattern Recognit.*, vol. 129, Sep. 2022, Art. no. 108742.

[19] A. B. Nassif, I. Shahin, I. Attili, M. Azzeh, and K. Shaalan, "Speech recognition using deep neural networks: A systematic review," *IEEE Access*, vol. 7, pp. 19143–19165, 2019.

[20] C. Pornprasit and C. Tantithamthavorn, "DeepLineDP: Towards a deep learning approach for line-level defect prediction," *IEEE Trans. Softw. Eng.*, early access, Jan. 21, 2022, doi: 10.1109/TSE.2022.3144348.

[21] C. Manjula and L. Florence, "Deep neural network based hybrid approach for software defect prediction using software metrics," *Cluster Comput.*, vol. 22, no. S4, pp. 9847–9863, Jan. 2018.

[22] N. Mandhan, D. K. Verma, and S. Kumar, "Analysis of approach for predicting software defect density using static metrics," in *Proc. Int. Conf. Comput., Commun. Autom.*, May 2015, pp. 880–886.

[23] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proc. 27th Int. Conf. Softw. Eng. (ICSE)*, 2005, pp. 284–292.

[24] D. Verma and S. Kumar, "Prediction of defect density for open source software using repository metrics," *J. Web Eng.*, vol. 16, no. 3, pp. 293–310, Dec. 2016.

[25] M. Sherriff, N. Nagappan, L. Williams, and M. Vouk, "Early estimation of defect density using an in-process Haskell metrics model," *ACM SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–6, Jul. 2005.

[26] M. Sherriff, L. Williams, and M. V. F. Abstract, "Using in-process metrics to predict defect density in Haskell programs," in *Proc. Int. Symp. Softw. Rel. Eng.*, St. Malo, France, 2004.

[27] A. Marchenko and P. Abrahamsson, "Predicting software defect density: A case study on automated static code analysis," in *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (Lecture Notes in Computer Science), vol. 4536. Berlin, Germany: Springer, 2007, pp. 137–140.

[28] O. Kutlubay, B. Turhan, and A. B. Bener, "A two-step model for defect density estimation," in *Proc. 33rd EUROMICRO Conf. Softw. Eng. Adv. Appl. (EUROMICRO)*, Aug. 2007, pp. 322–329.

[29] K. Sunint Khalsa, "A fuzzified approach for the prediction of fault proneness and defect density," in *Proc. World Congr. Eng.*, vol. 1, 2009, pp. 218–223.

[30] P. Knab, M. Pinzger, and A. Bernstein, "Predicting defect densities in source code files with decision tree learners," in *Proc. Int. Workshop Mining Softw. Repositories (MSR)*, 2006, pp. 119–125.

[31] H. B. Yadav and D. K. Yadav, "A fuzzy logic based approach for phase-wise software defects prediction using software metrics," *Inf. Softw. Technol.*, vol. 63, pp. 44–57, Jul. 2015.

[32] T. Hoang, H. Khanh Dam, Y. Kamei, D. Lo, and N. Ubayashi, "DeepJIT: An end-to-end deep learning framework for just-in-time defect prediction," in *Proc. IEEE/ACM 16th Int. Conf. Mining Softw. Repositories (MSR)*, May 2019, pp. 34–45.

[33] D. Chen, X. Chen, H. Li, J. Xie, and Y. Mu, "DeepCPDP: Deep learning based cross-project defect prediction," *IEEE Access*, vol. 7, pp. 184832–184848, 2019.

[34] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur.*, Aug. 2015, pp. 17–26.

[35] G. Zhao and J. Huang, "DeepSim: Deep learning code functional similarity," in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Oct. 2018, pp. 141–151.

[36] S. Ma, Y. Liu, W.-C. Lee, X. Zhang, and A. Grama, "MODE: Automated neural network model debugging via state differential analysis and input selection," in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Oct. 2018, pp. 175–186.

[37] J. Guo, J. Cheng, and J. Cleland-Huang, "Semantically enhanced software traceability using deep learning techniques," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. (ICSE)*, May 2017, pp. 3–14.

[38] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *Proc. 40th Int. Conf. Softw. Eng.*, May 2018, pp. 933–944.

[39] M. White, C. Vendome, M. Linares-Vasquez, and D. Poshyvanyk, "Toward deep learning software repositories," in *Proc. IEEE/ACM 12th Work. Conf. Mining Softw. Repositories*, May 2015, pp. 334–345.

[40] X. Huo and M. Li, "Enhancing the unified features to locate buggy files by exploiting the sequential nature of source code," in *Proc. 26th Int. Joint Conf. Artif. Intell.*, 2017, pp. 1909–1915.

[41] S. Jana, Y. Tian, K. Pei, and B. Ray, "DeepTest: Automated testing of deep-neural-network-driven autonomous cars," in *Proc. 40th Int. Conf. Softw. Eng.*, Aug. 2018, pp. 303–314.

[42] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and N. Tien Nguyen, "Combining deep learning with information retrieval to localize buggy files for bug reports (N)," in *Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2015, pp. 476–481.

[43] Y. Ma, G. Luo, X. Zeng, and A. Chen, "Transfer learning for cross-company software defect prediction," *Inf. Softw. Technol.*, vol. 54, no. 3, pp. 248–256, 2012.

[44] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: A benchmark and an extensive comparison," *Empir. Softw. Eng.*, vol. 17, nos. 4–5, pp. 531–577, 2011.

[45] F. Peters and T. Menzies, "Privacy and utility for defect prediction: Experiments with MORPH," in *Proc. 34th Int. Conf. Softw. Eng. (ICSE)*, Jun. 2012, pp. 189–199.

[46] M. Shepperd and S. MacDonell, "Evaluating prediction systems in software project estimation," *Inf. Softw. Technol.*, vol. 54, no. 8, pp. 820–827, Aug. 2012.

[47] M. Azzeh, A. B. Nassif, and S. Banitaan, "An application of classification and class decomposition to use case point estimation method," in *Proc. IEEE 14th Int. Conf. Mach. Learn. Appl. (ICMLA)*, Dec. 2015, pp. 1268–1271.

[48] Q. Song, M. Shepperd, and C. Mair, "Using grey relational analysis to predict software effort with small data sets," in *Proc. 11th IEEE Int. Softw. Metrics Symp. (METRICS)*, 2005, pp. 321–330.

[49] E. Kocaguneli and T. Menzies, "Software effort models should be assessed via leave-one-out validation," *J. Syst. Softw.*, vol. 86, no. 7, pp. 1879–1890, Jul. 2013.

[50] T. Menzies, Z. Chen, J. Hihn, and K. Lum, "Selecting best practices for effort estimation," *IEEE Trans. Softw. Eng.*, vol. 32, no. 11, pp. 883–895, Nov. 2006.

[51] J. Nam, W. Fu, S. Kim, T. Menzies, and L. Tan, "Heterogeneous defect prediction," *IEEE Trans. Softw. Eng.*, vol. 44, no. 9, pp. 874–896, Sep. 2018.

FIRAS ALGHANIM (Member, IEEE) received the B.Sc. degree in computer science from Princess Sumaya University for Technology (PSUT), Jordan, in 1996, and the Ph.D. degree in computer science from Durham University, U.K., in 2013, with focus on software engineering (HCI).

He was the Chair of the Department of Software Engineering, PSUT, from 2019 to 2021, where he is currently the Vice Dean of the School of Computing Sciences. His research interests include software engineering in general, software safety and security, and mobile computing. He has a very good experience in the industry in technical, managerial, and operational positions. He has been a member of ACM, since 2018, and the coach of top regional teams in ICPC and IEEEXtreme programming contests.

MOHAMMAD AZZEH received the M.Sc. degree in software engineering from the University of the West of England, U.K., and the Ph.D. degree in computing from the University of Bradford, U.K. He is currently a Professor of data science at Princess Sumaya University for Technology. He is also working as a Faculty Staff Member with the Data Science Department. He published over 40 research papers in reputable journals and conferences, such as *IET Software*, *Journal of Software: Evolution and Process*, *Empirical Software Engineering*, *Applied Soft Computing*, *Journal of Systems and Software*, *Science of Computer Programming*, and *Software Quality Journal*. His research interests include data science, machine learning, data mining, empirical software engineering, and mining software repositories. He was the Conference Chair of the 7th and 8th International Conference of Computer Science and Information Technology (2016 and 2018). He co-organized and co-chaired several special sessions/workshops: Computational Intelligence Applications in Software Engineering (CIASE 2013) at the 3rd IEEE International Conference on Communications and Information Technology (ICCIT 2013) and the Machine Learning for Predictive Models (MLPM 2013, 2014) at the IEEE International Conference on Machine Learning and Applications (ICMLA 2013 to 2021). He was also the Publicity Chair of CSIT 2014 Conference. He is an invited referee for high-quality journals and a PC member of international conferences. He was a Guest Editor of the *Journal of Neural Computing and Applications* (Springer).

AMMAR EL-HASSAN received the Ph.D. degree from Surrey University, U.K., in 2001.

He has extensive industrial experience in the U.K. academic and financial industries. He has been the College Dean as well as the Department Chair in Saudi Arabia for over ten years. He is a quality assurance and academic accreditation expert; creating a synergy between his commercial IT development experience, advances in education and research. Promoting the value of information from QA data, he has developed data management systems for accreditation and published research in machine learning for closing the loop in the learning process. He is currently the Head of Digital Transformation at the PSUT, where he is also an Assistant Professor at the King Hussein College of Computing Sciences, where he is teaching programming, database and data science courses at degree and master's levels as well as leading the college efforts in local, regional and international accreditation, this is in addition to supervision of master's students in computing and data science.

HAZEM QATTOUS received the B.Sc. degree from the Department of Computer Science, Applied Science Private University (ASU), in 2004, the M.Sc. degree in software engineering from the University of the West of England (UWE), Bristol, U.K., and the Ph.D. degree in meta-CASE tools and programming by example from Glasgow University, Glasgow, U.K., in 2011, with a full sponsorship. He worked as an Assistant Professor and an Associate Professor at ASU for nine years. During this period, he headed the Computer Science Department for two years and worked as the Vice Dean for one year. He is currently an Assistant Professor at the Department of Software Engineering, Princess Sumya University for Technology (PSUT).