Information and Software Technology 114 (2019) 204–216

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

# Improving defect prediction with deep forest

Tianchi Zhou[a], Xiaobing Sun[a,d,*], Xin Xia[b], Bin Li[a], Xiang Chen[c]

[a] School of Information Engineering, Yangzhou University, Yangzhou, China
[b] Faculty of Information Technology, Monash University, Melbourne, Australia
[c] School of Software, Northwestern Polytechnical University, Xi'an, China
[d] State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China

ARTICLE INFO

*Keywords:*
Software defect prediction
Deep forest
Cascade strategy
Empirical evaluation

ABSTRACT

*Context:* Software defect prediction is important to ensure the quality of software. Nowadays, many supervised learning techniques have been applied to identify defective instances (*e.g.*, methods, classes, and modules).
*Objective:* However, the performance of these supervised learning techniques are still far from satisfactory, and it will be important to design more advanced techniques to improve the performance of defect prediction models.
*Method:* We propose a new deep forest model to build the defect prediction model (*DPDF*). This model can identify more important defect features by using a new cascade strategy, which transforms random forest classifiers into a layer-by-layer structure. This design takes full advantage of ensemble learning and deep learning.
*Results:* We evaluate our approach on 25 open source projects from four public datasets (*i.e.*, NASA, PROMISE, AEEEM and Relink). Experimental results show that our approach increases AUC value by 5% compared with the best traditional machine learning algorithms.
*Conclusion:* The deep strategy in *DPDF* is effective for software defect prediction.

## 1. Introduction

Due to the complexity of modern software development, defects are unavoidable. Software projects that contain defects will have unintended consequences when they are deployed, which can result in huge losses to enterprises or even threaten the safety of people's lives [1–3]. At present, more than 80% of the cost during software development and maintenance is used in fixing defects [4–7]. If these defects can be detected in the software development cycle in the early phase, the cost would be greatly reduced. Therefore, many studies try to build predictive models for defect prediction to help developers detect possible defects in advance.

Unfortunately, there are still some problems for existing defect prediction models, such as the unsatisfactory performance of classifiers [8]. In order to solve these problems, a number of machine learning techniques have been proposed to predict software defects [9–15], e.g., Naive Bayes (NB) [16], Logistic Regression (LR) [17], Random Forest (RF) [18] and Support Vector Machine (SVM) [19]. However, these models are still far from satisfactory [20,21].

Recently, as deep learning has achieved good results in other areas (*e.g.*, image processing [22], speech recognition [23]), some prior work has attempted to make use of deep learning algorithms on defect prediction [24,25] and these algorithms have been proved to be promising in identifying defects. However, some problems may limit the applications of deep neural networks in software defect tasks. For example, deep neural networks need a huge amount of data to train the model, but there are not sufficient defective data in many software systems at present. Also, it is well known that the performance of these defect prediction models largely depends on the precise tuning of the hyper-parameters [26], and deep learning models normally have a number of hyper-parameters which are hard to determine. Moreover, the structure of deep neural networks has detached from human physiology and the near infinite combination makes it hard to recognize and explain.

Considering the above difficulties, recently, a new deep learning framework — deep forest model, called gcForest, has been proposed as an alternative of deep neural networks [26]. Similar to the deep neural networks, gcForest has multi-layer structure, and each layer contains two random forests and two completely-random tree forests instead of neurons in deep neural networks. What's more, gcForest can achieve satisfactory results in many tasks without too much tuning skills. However, two drawbacks of gcForest may limit its application on software defect prediction. First, the number of features extracted for defect prediction is limited (e.g., fewer than 40 in our datasets), but gcForest's multi-grained scanning strategy will be prone to generate many high-dimensional features which may not have special significance. The other is that the cascade multi-layer ensemble of random forests encourages the diversity of input but does not take into account differences of input, especially in the small-scale datasets, which makes gcForest sensitive and unrobust.

* Corresponding author.
*E-mail address:* xbsun@yzu.edu.cn (X. Sun).

https://doi.org/10.1016/j.infsof.2019.07.003
Received 16 July 2018; Received in revised form 5 June 2019; Accepted 5 July 2019
Available online 5 July 2019

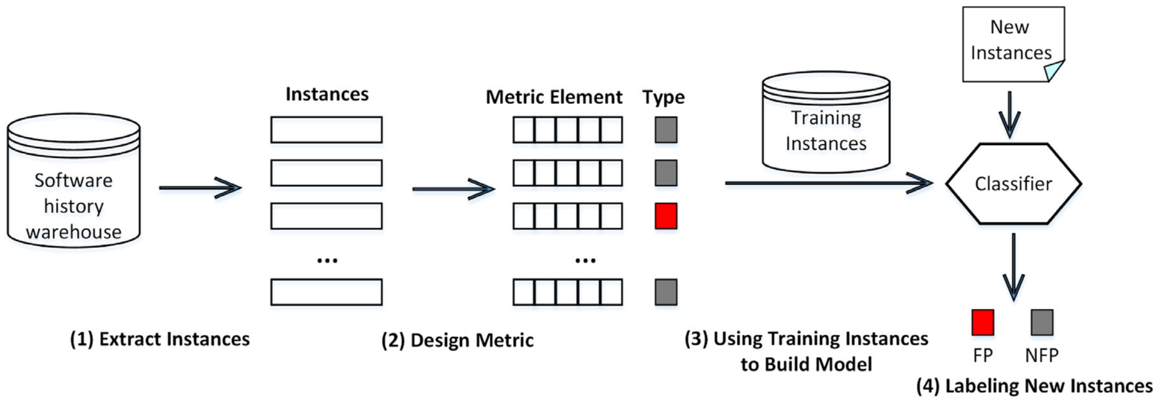

**Fig. 1.** The process of software defect prediction.

Therefore, to take advantage of gcForest, it is important to modify it to work better for the software defect prediction task.

In this paper, inspired by gcForest, we propose a new deep forest model by adopting a new cascade strategy, i.e., we adopt the idea of ensemble learning and deep learning. In detail, we transform the forest classifiers into a layer-by-layer structure and combine random forest classifiers in each layer, namely the defect prediction based on deep forest (*DPDF*), to perform the task of software defect classification. The main idea of our *DPDF* model is to use layer-by-layer training method to take the advantage of deep learning and combine random forest classifiers to encourage diversity and differences of input and increase the robustness of the model to identify more important defect features. Moreover, during the prediction process, we use the *z*-score [27] to standardize the feature value in the same magnitude that can be analyzed and evaluated synthetically.

In order to evaluate the effectiveness of our approach, we use four widely-used public software defect prediction datasets (*i.e.*, NASA, PROMISE, AEEEM and Relink) which include 25 projects to conduct our study. We compare our approach with six baselines, *i.e.*, Deeper, the approach proposed by Yang et al. [24]; gcForest, the approach proposed by Zhou et al. [26]; random forest, one of the basic structure of our approach and one of the state-of-the-art defect prediction approaches proposed by Ghotra et al. [21]; and three classic classifiers. Our major contributions are presented as follows:

- To our best knowledge, we are the first to consider building a deep forest model to predict software defects. Empirical results show that our approach achieves the best results on most of the projects, compared with the state-of-the-art supervised learning techniques. Moreover, our *DPDF* improves the AUC value by 8% on average compared with a model with deep learning method (DBN).
- We show that it is effective to apply a cascade strategy for defect prediction. Empirical results show that our approach gets better performance than the basic structure of our model on the NASA and PROMISE projects, which improves the AUC value by 7% on average compared with the model without the cascade strategy.

This paper is organized as follows: Section 2 shows the background. In Section 3, we describe details of using our *DPDF* model for defect prediction. The empirical setup is shown in Section 4. Section 5 discusses our empirical results. We discuss the parameters of the model, other evaluation measures and threats to validity in Section 6. We present the related work in Section 7. Finally, we conclude our study and discuss possible improvements for future work in Section 8.

## 2. Background

In this section, we mainly present the background of software defect prediction techniques and the deep forest used in our approach for defect prediction.

### 2.1. Software defect prediction

Software defect prediction is an important research problem in the field of software engineering. Fig. 1 shows the process of software defect prediction, including four steps:

1. Extract program modules/files/classes by mining software historical repositories, and the program modules/files/classes are then labeled as defect-proneness or not.
2. Extract features that are related to software defects by analyzing software code or the development process. Then, these features (e.g., Halstead features [28], McCabe features [29], and CK features [30]) are used to measure the defect-proneness of each program module/file/class. In Fig. 1, we use gray to represent the non defect-proneness (NFP) module while using red to represent the defect-proneness (FP) module.
3. Construct defect prediction model by training the instances with the corresponding features. In this step, machine learning algorithms (e.g., Naive Bayes [31], Support vector machine [32], Random forest [33], Logistic regression [17]) are widely used.
4. Use the defect prediction model to predict the unlabeled program modules/files/classes, i.e., classify them as either defective or not.

Software defect prediction can be used for within-project defect prediction (WPDP) and cross-project defect prediction (CPDP). In our work, we focus on the within-project defect prediction, which means that the training datasets and testing datasets belong to the same project.

### 2.2. Deep forest

Recently, a deep forest model called gcForest has been proposed as an alternative to deep neural networks [26]. Similar to deep neural networks, gcForest has multi-layer structure and each layer contains many forests. The basic structure of a forest is shown in Fig. 2. The input is a feature vector *X*, and the output is the class vector of *X* based on the decision tree judgment in the forest. In Fig. 2, the red path represents the decision tree's decision-making process. Inspired by the deep neural network, gcForest consists of two ensemble components. The first one is the multi-grained scanning operation, which adopts sliding window structure to scan local context from high-dimensionality to learn representations of input data according to different random forests. The second one is the cascade multi-layer ensemble of random forests, which learns more discriminated representations under supervisor of input representations at each layer, thus gives more accurate predictions according to ensemble of random forests. Unlike traditional deep neural networks, which need a large amount of computational resources to train, deep forest is much easier to train. Moreover, the deep forest can achieve good results in many tasks with the default parameters [26].

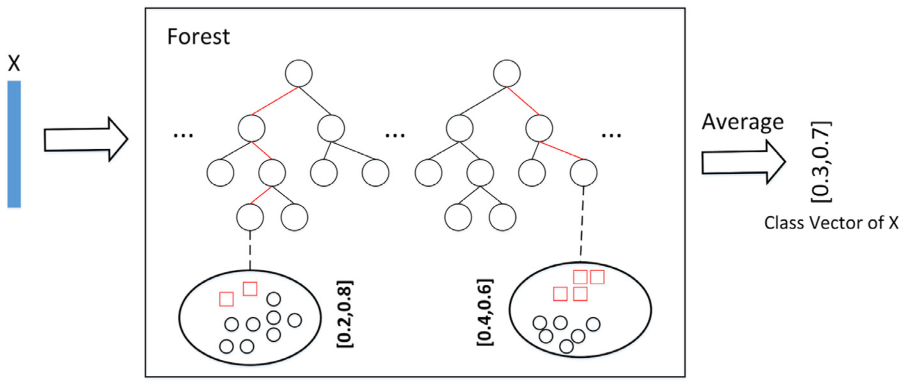

**Fig. 2.** The basic structure of a forest.

Compared with the random forest, the main differences of gcForest are as follows:

- GcForest is a layer-by-layer structure and each layer is an ensemble of decision tree forests, where each level of cascade receives feature information processed by its preceding level, and outputs its processing result to the next level [26]. So gcForest is an ensemble of ensembles. Meanwhile, random forest is an ensemble of decision trees.
- GcForest has a strong ability to handle features by using multi-grained scanning [26], which can generate some new features. Meanwhile, random forest is only processed for the original input features.

For software defect prediction, there are some problems in construction of the gcForest. As we know, the source code is special and it is a collection of abstract logical symbols. So it is usually difficult to extract the features of code and the features extracted from metrics are usually with low dimension. However, gcForest produces a series of high-dimensional features by multi-grained scanning. These high-dimensional features may not have a specific meaning, which is likely to reduce the classification performance. In our work, to address the limitation of deep forest for defect prediction, we propose a new cascade forest structure to fit for the software defect prediction task.

## 3. Proposed approach

In this section, we elaborate our proposed *DPDF*, a tree network based software defect prediction using the cascade forest structure. As Fig. 3 shows, the whole process of *DPDF* is illustrated as follows:

1. We use the *z*-score to standardize each feature in our defect datasets by referring to the work of Zhang et al. [34].
2. We use the cascade strategy on random forest classifier. This means that we use the cascade forest structure to perform representation learning by processing the raw defect features through the layer-by-layer structure, which makes our defect prediction model go deep. Some studies suppose that traditional machine learning may exhibit great potentials if being able to go deep [26]. At the last layer, our *DPDF* outputs the results of defect prediction.

### 3.1. Process of defect prediction

Fig. 3 shows the overall procedure about the *DPDF* for defect prediction. For example, in one project, there are $N$ defect features. So the original input has $N$ defect features. There are $n$ instances and two classes (FP class and NFP class) in the training datasets. For a cascade forest, a $N$-dimensional feature vector consisting of $N$ defect features will be used to train the 1st-grade of the cascade forest. After that, each $N$-dimensional feature vector is converted to a 2-dimensional class vector in each forest. Then, all of the 2-dimensional class vectors (there are $M$ forests in each layer) and the original $N$-dimensional feature vector are combined with a ($N$+2$M$)-dimensional feature vector, which will be used to train the next layer of cascade forest and the propagation of cascade is automatically terminated once there is no significant performance improvement.

Generally, given a defective instance, the first step is to measure the defective instance by using the software metric. Next, we use the *z*-score to standardize the metric. After that, the feature representation is processed by the cascade forests, and the final prediction is obtained by averaging the $M$ 2-dimensional vectors, and the maximum value of the final prediction is chosen as the prediction result.

### 3.2. Standardize defect prediction datasets

For software defect prediction, standardizing the defect prediction datasets is necessary because the software features often have different orders of magnitude. In this paper, we also normalize the data, mainly referring to the work of Zhang et al., and adopting the *z*-score method[1] [34]. The *z*-score can normalize the feature to obtain a mean value of zero and a variance of one.

*DPDF* uses the *z*-score to standard each feature. We use $X_j$ to denote a vector of values of the $j$th feature in a project. Then $X_j = [a_{1j}, \cdots, a_{nj}]^T$, where $n$ means the number of instances in the project, and $a_{ij}$ is the value of the $j$th feature of the $i$th instance. In *z*-score, the $X_j$ is processed as

$$\tilde{X}_j = \frac{X_j - \bar{X}_j}{S_j}$$

where $\bar{X}_j$ is the average value of $X_j$ and $S_j$ is the standard deviation of $X_j$.

### 3.3. The cascade structure

Representation learning can help extract useful information from features when building predictors, which mostly relies on the layer-by-layer processing of raw features [35]. Deep forest consists of the cascade forest structure which is used to replace the representation learning in deep neural networks [26]. In our work, the proposed cascade strategy is used to combine random forest classifiers into a layer-by-layer structure.

Our cascade forest structure uses the decision tree to learn defect features instead of learning hidden variables based on the complex forward-propagation algorithm and back-propagation algorithm in deep neural network. The layer-by-layer cascade forest structure enables the traditional forest to effectively go deep. Moreover, each layer contains four random forests, which can increase the diversity of polymerization and model robustness by taking full advantage of ensemble learning. The parameters of these forests use default values except the number of decision trees. Users can find random forests' default settings on the

[1] We also tried to use the min-max normalization for 25 data sets on the Naive Bayes method and find that the effect of AUC is consistent.

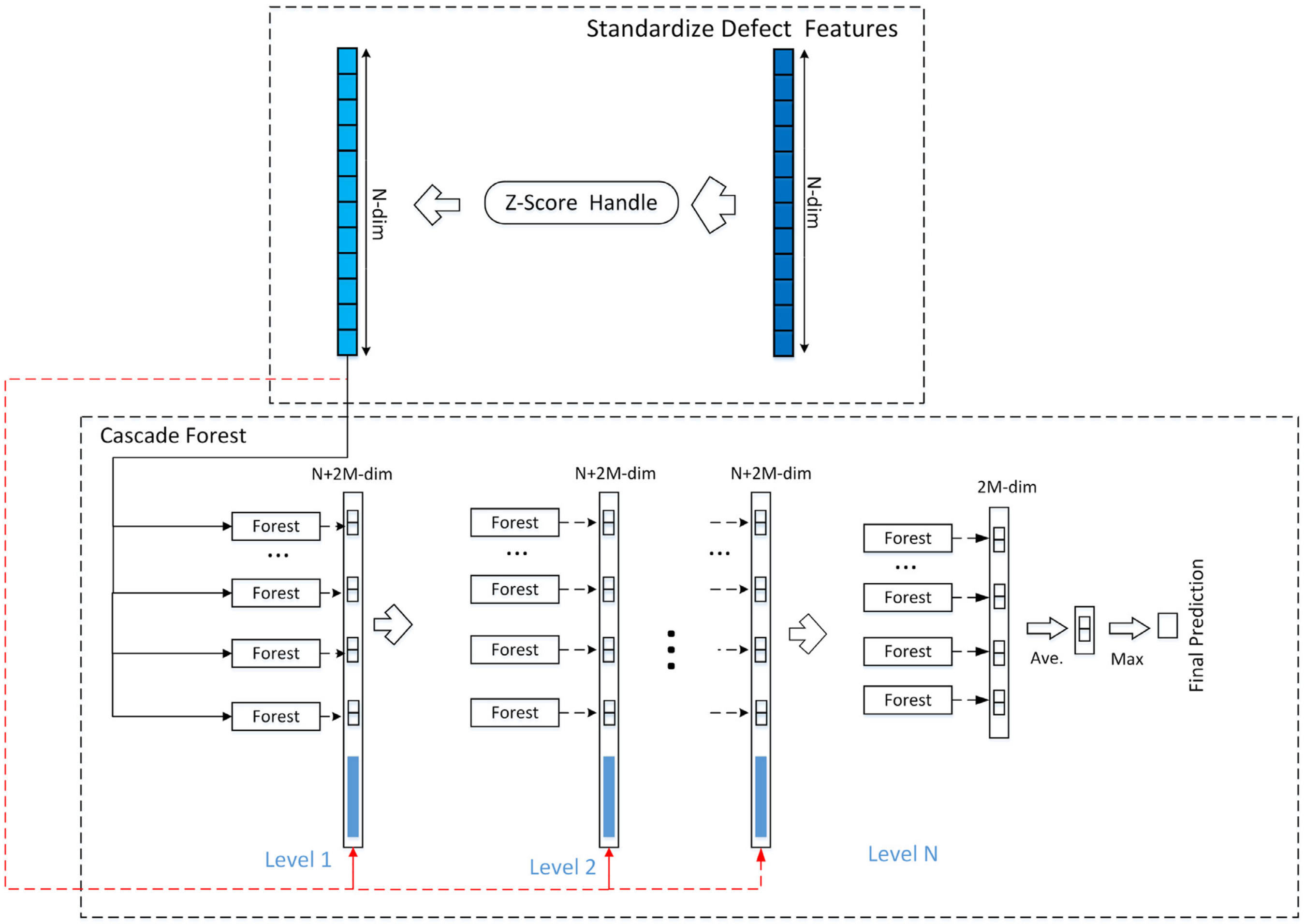

**Fig. 3.** The overall architecture of our proposed approach.

scikit-learn official website.[2] The only parameter the user needs to decide is the number of decision trees in each forest. We will discuss this parameter in the later section of this paper. In our model, each random forest will randomly select $\sqrt{n}$ number of features as candidate ($n$ is the number of input features) and choose the one with the best *gini* value for split. The estimated class distribution forms a class vector, which is then concatenated with the original feature vector for input to the next level. For example, suppose there are two classes, then each of the random forest will produce a two-dimensional class vector. So, the next level will receive 8 ($= 2 \times 4$) augmented features. The new generated features and the original features will feed to all forests in the next layer.

Fig. 3 shows the structure of our cascade forest model. Given an instance, each forest can produce an estimation of class distribution by counting the percentage of different classes of training examples at the leaf node. The performance of the whole cascade is estimated on the testing set after expanding a new layer, and the propagation of cascade will be automatically terminated once there is no significant performance improvement. Finally, the cascade forests can be used to predict the new defective instances.

## 4. Experiment setup

In this section, we describe the details of setup in our empirical study, which is used to evaluate the effectiveness of our approach for software defect prediction.

### 4.1. Datasets

In order to evaluate the effectiveness of our approach, we choose four open datasets in NASA, PROMISE, AEEEM and Relink which are often used to evaluate defect prediction [21,36,37]. We obtain these datasets in tera-PROMISE.[3] A brief description of these datasets and the used software metrics on each dataset are as follows:

**The NASA dataset** was collected by the NASA Metrics Data Program. In 2013, Shepperd et al. have cleaned up the repeated and inconsistent data in NASA defect prediction datasets [38]. This improved datasets can be achieved in the PROMISE repository. So in our study, we used the cleaned NASA dataset. In the NASA datasets, they use Halstead and McCabe metrics for each instance. There are nearly 40 features, including the number of unique operators (MU1), number of unique operands (MU2), total occurrences of operators (N1), total occurrences of operands (N2), Lines of Code (LOC), etc. In the NASA project, each project uses different features. The specific number of features can be found in Table 1.

**The PROMISE dataset** was developed by Jureczko et al. [39], which contains various open source Java projects. Like the NASA dataset, we also obtain the dataset from tera-PROMISE Home. In PROMISE, the projects have different metrics, including Lines of Code (LOC), Response for a Class (RFC), Average McCabe, Average Method Complexity (AMC), Coupling Between Object classes (CBO), etc. In PROMISE project, all

[2] http://scikit-learn.org/stable/.

[3] http://openscience.us/repo/defect/.

**Table 1**
Statistics of the datasets.

| Dataset | Project | Instances | Defects | % of Defects | Features |
|---|---|---|---|---|---|
| NASA | JM1 | 7782 | 1672 | 21.5% | 21 |
| | MC1 | 1988 | 46 | 2.3% | 38 |
| | MC2 | 125 | 44 | 35.2% | 39 |
| | MW1 | 253 | 27 | 10.7% | 37 |
| | PC1 | 705 | 61 | 8.7% | 37 |
| | PC2 | 745 | 16 | 2.1% | 36 |
| | PC3 | 1077 | 134 | 12.4% | 37 |
| | PC4 | 1287 | 177 | 13.8% | 37 |
| | PC5 | 1711 | 471 | 27.5% | 38 |
| PROMISE | Xalan v2.6 | 885 | 411 | 46.4% | 20 |
| | Ant v1.7 | 745 | 166 | 22.3% | 20 |
| | Camel v1.6 | 965 | 188 | 19.5% | 20 |
| | Jedit v4.0 | 306 | 75 | 24.5% | 20 |
| | Log4j v1.0 | 135 | 34 | 25.2% | 20 |
| | Lucene v2.4 | 340 | 203 | 59.7% | 20 |
| | Poi v3.0 | 442 | 281 | 63.6% | 20 |
| | Tomcat v6.0 | 858 | 77 | 9.0% | 20 |
| AEEEM | LC | 399 | 39 | 9.3% | 61 |
| | JDT | 997 | 206 | 20.7% | 61 |
| | PDE | 1492 | 209 | 14.0% | 61 |
| | EQ | 325 | 129 | 39.6% | 61 |
| | ML | 1862 | 245 | 13.2% | 61 |
| Relink | Apache | 194 | 98 | 50.5% | 26 |
| | Safe | 56 | 22 | 39.3% | 26 |
| | Zxing | 399 | 118 | 29.6% | 26 |

projects use the same features, and the number of features can be found in Table 1.

**The AEEEM dataset** was collected by D'Ambros et al. [40], which comes from Eclipse and Apache. In AEEEM, each program has 61 different metrics, which combines many classic metrics. Specifically, MOSER (file-level change metrics), CK, OO (object-oriented metrics), LOC, HCM(history of complexity metric), LDHH (source code entropy metric) and other metrics are all in AEEEM.

**The Relink dataset** was developed by Wu et al. [41]. Relink dataset has 26 complexity metrics, which have been widely used in software defect prediction.

Finally, in order to ensure the diversity of the used data, the number of project instances ranges from 56 to 7782. Moreover, in order to test our model performance on unbalanced datasets, we choose projects with different proportions of defects, ranging from 2.1% to 63.6%. Table 1 shows the statistics about the datasets. The second column shows these 25 projects in the datasets. The instances and their defective instances are shown in the third and fourth column, respectively, and an instance is a source code file. The fifth column represents the percentage of defective instances. The number of features in each project is shown in the last column.

## *4.2. Evaluation measure*

There are lots of evaluation measures used to evaluate machine learning classifiers, such as precision, recall, accuracy, *f*-measure and Area Under the receiver operating characteristic Curve (AUC). Among these measures, AUC is widely used in many software engineering studies [20,42,43]. AUC is the area under the receiver operating characteristic curve. This curve is plotted in a two-dimensional space with false positive rate as *x*-coordinate and the true positive rate (recall) as *y*-coordinate.

There are three reasons why we chose AUC as our performance measure. (1) Unlike other measures which need a cut-off value on the predicted probability of defect proneness, AUC is a threshold independent measure [44]. The threshold indicates a likelihood threshold for determining whether the instance is classified as positive or negative. Other performance measures (*e.g.*, precision, recall, accuracy, and *f*-measure) rely on the selected threshold and are usually set as 0.5. However, for some cases (*e.g.*, the class imbalance case), choosing a threshold is a challenging task, which may impact the performance evaluation. AUC can effectively avoid the threshold setting problem because AUC measures the classification performance across all thresholds (i.e., from 0 to 1). In recent work, Tantithamthavorn and Hassan recommend to use threshold-independent measures (e.g., AUC) to replace threshold-dependent measures (e.g., *f*-measure) because *f*-measure is sensitive to the selected threshold [45]. (2) AUC is more robust towards class distribution than other measures (*e.g.*, precision, recall, accuracy, and *f*-measure) [20]. Other performance measures are highly affected by class distribution. This might make it difficult to fairly compare two models by using precision, recall, accuracy and *f*-measure in these cases [42,46]. Unlike other measures, the AUC measure is less insensitive to class distribution. (3) What's more, AUC has a statistical interpretation [20]. In our work, the AUC is equivalent to the probability that a randomly chosen defect-prone instance is ranked higher than a randomly chosen non defect-prone instance. Since our motivation is to improve the performance of software defect prediction, the AUC is an appropriate measure to evaluate the performance of our approach and baseline methods.

As mentioned above, the AUC is equivalent to the probability that a randomly chosen defect-prone instance is ranked higher than a randomly chosen non defect-prone instance [47]. A higher AUC value means that the performance of the machine learning classifier is much better. If the AUC value is equal to 0.5, this means that the classifier is the same as random guess. In general, Gorunescu [48] suggests the following guideline to interpret the AUC value: 0.90 to 1.00 as an excellent prediction, 0.80 to 0.90 as a good prediction, 0.70 to 0.80 as a fair prediction, 0.60 to 0.70 as a poor prediction, and 0.50 to 0.60 as a failed prediction. So when the AUC value is under 0.5, the performance of the classifier is worse than the random guess, which is a failed prediction.

## *4.3. Baseline methods*

To compare the effectiveness of our *DPDF* with other methods for defect prediction, we compare our classifiers with six baseline classifiers, *i.e.*, gcForest, Deep belief networks(DBN), random forest(RF), naive Bayes (NB), logistic regression(LR) and Support Vector Machine(SVM). Some of them have been commonly applied to build defect prediction models [21,24,31–33,49–51]. In a previous empirical study, Ghotra and Hassan have proved that RF is a state-of-the-art defect classifiers according to the double SCOTT-KNOTT test compared with 31 different classifiers on the clean NASA datasets [21]. Moreover, for defect prediction, Yang et al. have proved that deep learning method named deep belief networks(DBN) can achieve strong performance [24]. When implementing DBN, we use the same network structure and parameter settings as in [24]. About other classical classifiers, these classifiers are selected from existing work, which were widely used for defect prediction [31–33,49–51]. In addition, the original deep forest called gcForest will also be used to compare with our method in order to test the effectiveness of our method based on the gcForest. What's more, all baseline methods use *z*-score to process the original data.

## *4.4. Research questions*

In software defect prediction, the proposed techniques mainly focused on the performance and stability of the classifier. In this paper, our method uses the cascade strategy on random forest. Thus, we are interested in whether our prediction model is better than the basic structure of our model (Random Forest). What's more, the training speed of the model should also be used as a measure. Hence, we focus on the following three research questions:

- **RQ1**: How does our *DPDF* perform in within-project defect prediction?
- **RQ2**: What are the benefits of cascade strategy?
- **RQ3**: How much time does it take for DPDF to run?

### 4.5. Setup

Inspired by gcForest, we set four forests ($M = 4$) in each layer and 500 decision trees in each forest. We will also discuss parameter settings in Section 5.

In RQ1, we evaluate the performance of all defect prediction models. It is necessary to divide the datasets into two parts. One part is used to train a model and the other one is applied to test the correctness of the model. To create the training and testing datasets, we apply a cross-validation way called two-fold cross-validation (i.e., a 50:50 random split), which has been widely used in the defect prediction task [34,42,52]. In the process of a two-fold cross-validation, the first step is to use the first half dataset ($A$) as training set and the other half dataset ($B$) as testing set. Then, the two sets are conversed, *i.e.*, the $B$ dataset as training set while the other half $A$ dataset as testing set. In order to ensure the reliability of our experimental results, we repeat the random splits 100 times, which means that there are a total of 200 assessments on each model. Finally, the average AUC value is calculated to evaluate the performance of each model.

In RQ2, we evaluate the effect of deep strategy on defect prediction in NASA and PROMISE. Our *DPDF* is a layer-by-layer structure by combining random forests. Hence, we only need to compare our *DPDF* with random forest classifier. In this experiment, we still use the 100 two-fold cross-validation method as mentioned in RQ1.

In RQ3, to obtain the running time, we use Intel Xeon CPU E5645 @ 2.40GHz to train our model, which contains 24 logical CPU cores. Each running time includes the time taken for data preprocessing, classifier training and testing. For each dataset, we use the 10 two-fold cross-validation method to collect the time and then average these time as the running time.

In our study, we conducted a non-parametric Mann–Whitney $U$ test at a confidence level of 95% to statistically analyze the defect prediction results. Mann–Whitney $U$ test is a non-parametric statistical method that it makes no assumptions about the distribution of the data. This statistic test has been widely used in defect prediction researches [15,53,54] because Mann–Whitney $U$ test does not demand that the two compared populations are of the same size and it can avoid the needs of the Bonferroni-Dunn test to offset the results of multiple comparisons [55]. So we also choose this test to evaluate all results. Then, we report the win/tie/loss (w/t/l) results of our approach compared with each baseline. “Win” means that the performance of our method is better than other method at a confidence level of 95%, meanwhile, “tie” means “equal”, and otherwise “lose”. This report is also widely used in previous studies [3,56–61].

In addition, we also used Cliffs delta ($\delta$) to check whether the differences between two methods are substantial, which is a nonparametric effect size test [62]. To measure the degree of differences in AUC results between our approach and baselines, it is necessary to calculate Cliffs delta. The range of Cliffs delta is $[-1, 1]$, where $-1$ or 1 means that all values in one method are smaller or larger than those of the other method, and 0 means that the measure in the two methods is similar. As Romano et al. suggested [62], the magnitude of the effect size is as follows: *negligible* (N, $|\delta| < 0.147$), *small* (S, $0.147 \leq |\delta| < 0.33$), *medium* (M, $0.33 \leq |\delta| < 0.474$) and *large* (L, $|\delta| \geq 0.474$). This test has been widely used in defect prediction studies [24,56,63–65].

## 5. Result analysis

In this section, we present the empirical results to answer the proposed three research questions.

### 5.1. RQ1: effectiveness of the DPDF

#### 5.1.1. Comparative

To the best of our knowledge, we are the first to use the deep forest model to predict software defects. Therefore, we first examine the performance of our *DPDF* for defect prediction. Since the four datasets use different metrics, we evaluate all defect prediction classifiers on each project individually, as shown in Fig. 4.

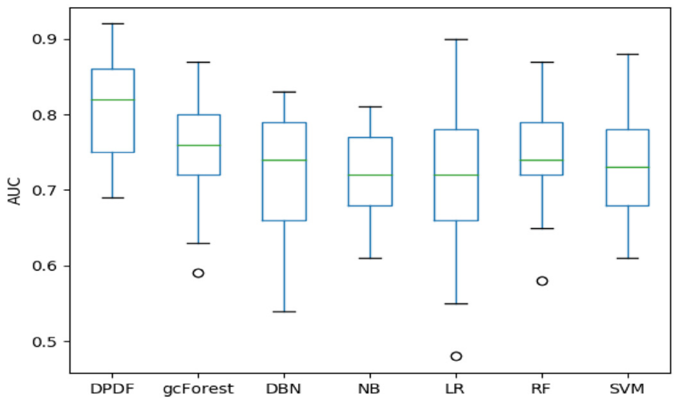

**Fig. 4.** The boxplots of AUC values of all classifiers.

As shown in Table 2, **For NASA datasets**, our defect prediction deep forest model (*DPDF*) is better than other approaches in all projects. Our *DPDF* improves the AUC value by 8% over the latest deep forest model(gcForest) on *PC1*. What’s more, our *DPDF* obtains 9% better performance than the state-of-the-art classifier (RF) suggested by Ghotra [21] on the NASA dataset.

**For PORMISE datasets**, as Table 2 shows, our model also achieves the best prediction results in all projects. In detail, our method improves the AUC value by 29% over the defect prediction model using deep learning proposed by Yang [24] on the *Xalan v2.4* project. What’s more, *DPDF* obtains 6% better performance than the best traditional classifier (RF) on the PROMISE dataset.

**For AEEEM datasets**, our model also achieves the best prediction in all projects. Specifically for ML projects, our model has improved performance by 18% compared to other deep learning methods (DBN).

**For Relink datasets**, our model also performs well overall. However, in some projects with few defect instances (*e.g.*, safe), our method still has some shortcomings.

On average, *DPDF* improves the AUC value by 5% on average compared with the best traditional machine learning algorithms (RF) for all the 25 projects from four datasets. Moreover, it is worth mentioning that our *DPDF* has achieved better results than the original deep forest model (gcForest) and deep learning model (DBN) in almost all experimental projects.

The reasons that *DPDF* achieves better results are as follows:

1. *Compared with traditional machine learning methods*: As shown in Fig. 4 and Table 2, these traditional machine learning classifiers (i.e., Naive Bayes, Logistic Regression, Support Vector Machines and Random Forest) have achieved similar prediction results in the selected 25 projects. This is because that traditional machine learning classifiers use defect features directly. Different from these methods, our *DPDF* can automatically learn more important defect features due to the proposed cascade structure.
2. *Compared with original deep forest methods*: From Fig. 4 and Table 2, the performance of gcForest is better than traditional defect prediction methods. However, software defect features are low-dimensional, and gcForest’s multi-grained scanning generates many irrelevant high-dimensional features that may interfere with subsequent training. Different from gcForest, our DPDF gives up multi-grained scanning and builds a new layer-by-layer structure, which can identify more important defect features effectively.

#### 5.1.2. Statistical significance test

The last second row in Table 2 shows the results of statistical significance test. We can notice that *DPDF* can statistically significantly improve the performance of baselines in most projects.

**Table 2**
The AUC results of all classifiers.

| Dataset | Project | DPDF | gcForest | DBN | NB | LR | RF | SVM |
|---|---|---|---|---|---|---|---|---|
| **NASA** | JW1 | **0.69** | 0.68 | 0.64 | 0.64 | 0.67 | 0.65 | 0.61 |
| | MC1 | **0.79** | 0.75 | 0.74 | 0.65 | 0.63 | 0.72 | 0.68 |
| | MC2 | **0.72** | 0.64 | 0.71 | 0.66 | 0.66 | 0.67 | 0.67 |
| | MW1 | **0.75** | 0.68 | 0.74 | 0.7 | 0.48 | 0.65 | 0.67 |
| | PC1 | **0.87** | 0.79 | 0.74 | 0.68 | 0.76 | 0.79 | 0.79 |
| | PC2 | **0.83** | 0.73 | 0.79 | 0.68 | 0.55 | 0.58 | 0.66 |
| | PC3 | **0.82** | 0.78 | 0.73 | 0.75 | 0.81 | 0.76 | 0.73 |
| | PC4 | **0.92** | 0.87 | 0.68 | 0.72 | 0.9 | 0.87 | 0.88 |
| | PC5 | **0.77** | 0.74 | 0.66 | 0.71 | 0.72 | 0.74 | 0.71 |
| **PROMISE** | Xalan v2.6 | **0.86** | 0.82 | 0.57 | 0.78 | 0.8 | 0.83 | 0.81 |
| | Ant v1.7 | **0.82** | 0.79 | 0.79 | 0.78 | 0.79 | 0.80 | 0.78 |
| | Camel v1.6 | **0.72** | 0.63 | 0.63 | 0.66 | 0.67 | 0.68 | 0.65 |
| | Jedit v4.0 | **0.82** | 0.8 | 0.75 | 0.76 | 0.72 | 0.77 | 0.77 |
| | Log4j v1.0 | **0.87** | 0.8 | 0.83 | 0.77 | 0.72 | 0.74 | 0.74 |
| | Lucene v2.4 | **0.79** | 0.72 | 0.75 | 0.71 | 0.76 | 0.73 | 0.75 |
| | Poi v3.0 | **0.88** | 0.85 | 0.81 | 0.8 | 0.79 | 0.86 | 0.81 |
| | Tomcat v6.0 | **0.89** | 0.84 | 0.82 | 0.77 | 0.78 | 0.77 | 0.73 |
| **AEEEM** | LC | **0.82** | 0.78 | 0.77 | 0.77 | 0.55 | 0.72 | 0.71 |
| | JDT | **0.86** | 0.79 | 0.79 | 0.81 | 0.78 | 0.83 | 0.84 |
| | PDE | **0.77** | 0.72 | 0.69 | 0.75 | 0.72 | 0.73 | 0.70 |
| | EQ | **0.85** | 0.81 | 0.81 | 0.79 | 0.66 | 0.84 | 0.82 |
| | ML | **0.82** | 0.76 | 0.64 | 0.72 | 0.70 | 0.76 | 0.78 |
| **Relink** | Apache | 0.75 | 0.73 | 0.54 | 0.74 | 0.70 | **0.76** | 0.76 |
| | Safe | 0.73 | 0.59 | **0.77** | 0.69 | 0.67 | 0.73 | 0.69 |
| | Zxing | **0.70** | 0.64 | 0.63 | 0.61 | 0.57 | 0.67 | 0.66 |
| Average | | **0.80** | 0.75 | 0.72 | 0.72 | 0.70 | 0.75 | 0.74 |
| win/tie/loss | | – | 25/0/0 | 23/1/1 | 25/0/0 | 25/0/0 | 24/1/0 | 24/1/0 |
| Cliffs delta | | – | 0.414(M) | 0.578(L) | 0.648(L) | 0.638(L) | 0.432(M) | 0.539(L) |

### 5.1.3. *Effect size test*

The results of Cliff's delta for effect size among the baselines and our approach are shown in the last row of Table 2. We can notice that the magnitudes between DPDF and baselines are mostly *large*. This means that our method is significantly better than baselines.

Based on the results, we conclude that our approach (*DPDF*) obtains the best performance compared with six baselines, and our improved deep forest can achieve better results in software defect prediction tasks than the previous deep forest (gcForest).

## 5.2. *RQ2: benefits of the cascade strategy*

In order to show the effect of our cascade strategy, it is necessary to compare our *DPDF* with random forest (RF) because the random forest classifier is the basic structure of our *DPDF*.

Fig. 5 shows the AUC results of *DPDF* and RF for defect prediction in NASA and PROMISE (total 17) projects, where the green color represents the values of *DPDF* while pink color represents that of RF. We can see that the performance of *DPDF* is better than that of random forest in all 17 projects, which means that our cascade strategy is useful for defect prediction. Specifically, *DPDF* improves AUC values by 7% on average compared with that of the random forest. Moreover, the best improvement for our model is on the *PC2* project, which increases the performance by 25%. The project *Lucene v2.4* has a ratio of defects of 2.1% with only 16 defective instances since such few defect instances is hard for traditional machine learning method to learn important defect features. So it is effective for our cascade strategy to handle feature relationships which can help provide more input information for the next layer input by abstracting and combining original features into the features with better recognition ability.

Based on the above results, we conclude that the cascade strategy is effective to increase the performance of traditional machine learning classifiers.

## 5.3. *RQ3: the running time of all models*

Table 3 reports the running time about all models on 25 datasets. From Table 3, it takes about 2 s for our approach to finish one running. Meanwhile, other deep models need more time to run. Although there

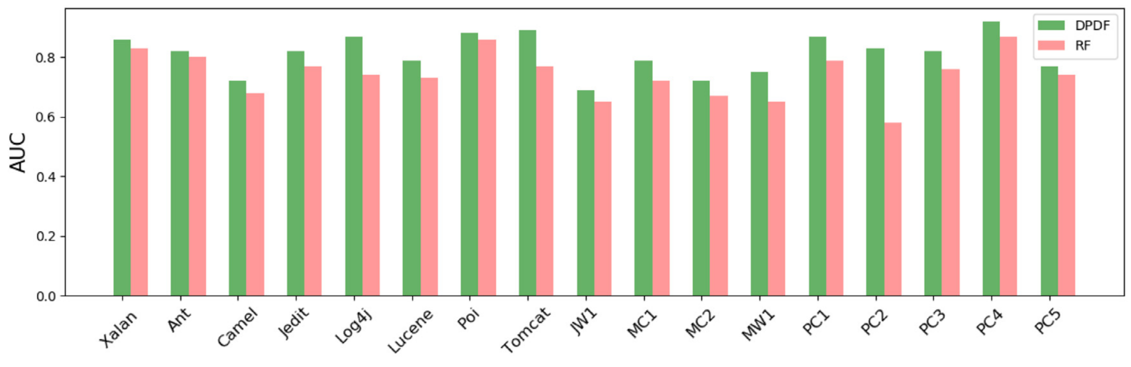

**Fig. 5.** The AUC results of *DPDF* and *Random Forest* for defect prediction in the 17 projects.

**Table 3**
The running time of all classifiers (in seconds (s)).

| Dataset | Project | DPDF | gcForest | DBN | NB | LR | RF | SVM |
|---|---|---|---|---|---|---|---|---|
| **NASA** | JW1 | 12.250 | 32.448 | 57.455 | 0.129 | 0.177 | 1.156 | 3.447 |
| | MC1 | 2.200 | 65.662 | 16.824 | 0.055 | 0.120 | 0.297 | 0.137 |
| | MC2 | 1.754 | 55.361 | 1.622 | 0.005 | 0.013 | 0.125 | 0.007 |
| | MW1 | 1.241 | 29.552 | 2.840 | 0.009 | 0.022 | 0.133 | 0.012 |
| | PC1 | 1.456 | 20.895 | 6.582 | 0.023 | 0.042 | 0.179 | 0.039 |
| | PC2 | 1.374 | 20.982 | 7.105 | 0.025 | 0.047 | 0.174 | 0.028 |
| | PC3 | 1.768 | 65.085 | 9.783 | 0.032 | 0.066 | 0.216 | 0.083 |
| | PC4 | 3.345 | 21.088 | 10.985 | 0.037 | 0.090 | 0.236 | 0.120 |
| | PC5 | 2.447 | 47.736 | 14.395 | 0.047 | 0.136 | 0.347 | 0.253 |
| **PROMISE** | Xalan v2.6 | 1.654 | 21.157 | 7.334 | 0.016 | 0.082 | 0.177 | 0.055 |
| | Ant v1.7 | 2.261 | 20.970 | 6.332 | 0.013 | 0.025 | 0.163 | 0.034 |
| | Camel v1.6 | 2.278 | 64.515 | 8.060 | 0.019 | 0.031 | 0.204 | 0.058 |
| | Jedit v4.0 | 1.288 | 47.039 | 2.818 | 0.007 | 0.018 | 0.143 | 0.011 |
| | Log4j v1.0 | 1.855 | 29.322 | 1.597 | 0.005 | 0.016 | 0.119 | 0.005 |
| | Lucene v2.4 | 1.857 | 29.609 | 3.135 | 0.006 | 0.019 | 0.139 | 0.014 |
| | Poi v3.0 | 1.282 | 20.903 | 3.879 | 0.009 | 0.02 | 0.145 | 0.018 |
| | Tomcat v6.0 | 1.537 | 29.707 | 7.109 | 0.016 | 0.029 | 0.166 | 0.038 |
| **AEEEM** | LC | 1.556 | 21.149 | 7.418 | 0.030 | 0.081 | 0.179 | 0.060 |
| | JDT | 1.761 | 58.246 | 10.138 | 0.046 | 0.093 | 0.239 | 0.100 |
| | PDE | 3.147 | 39.111 | 14.539 | 0.067 | 0.125 | 0.321 | 0.198 |
| | EQ | 1.231 | 55.886 | 3.839 | 0.015 | 0.040 | 0.177 | 0.026 |
| | ML | 2.460 | 29.562 | 17.936 | 0.083 | 0.139 | 0.346 | 0.298 |
| **Relink** | Apache | 2.343 | 47.024 | 2.062 | 0.006 | 0.018 | 0.137 | 0.010 |
| | Safe | 1.115 | 38.198 | 1.013 | 0.004 | 0.011 | 0.131 | 0.004 |
| | Zxing | 1.316 | 47.135 | 3.732 | 0.012 | 0.023 | 0.150 | 0.019 |

is no way to compare running time with traditional machine learning classifiers, we believe that 2 s is an acceptable cost.

## 6. Discussions

### 6.1. Performance of DPDF under different parameter settings

In this section, we discuss the performance of our *DPDF* under different parameter settings. Because many researchers suppose that the performance of deep learning models are affected by large and complex parameter settings, which depends on the skills of parameter tuning. However, for our *DPDF*, the users only need to decide how many decision trees are in each forest. For the other parameter settings about each forest, we adopted the default values and these default parameters can be easily obtained in the official scikit-learn documentation.

To show this, we vary the values of this parameter and conduct experiments on project *PC2, PC4*, and *Jedit v4.0*, respectively. Fig. 6 shows the AUC of *DPDF* under different numbers of decision trees. The first point represents the AUC value of *DPDF* under 100 decision trees in each forest and the last point represents the AUC value of *DPDF* under 1000 decision trees in each forest. We can see that the optimal number of decision trees is 500, where the three curves generally reach the peak. What's more, considering that the larger number of decision trees is, the more running time we will take, so we choose the value of this parameter as 500.

Moreover, we can see that the performance of *DPDF* under different parameter settings is relatively stable, which means that the users can obtain satisfactory defect prediction effect by using our default setting.

Based on the results, we show that our *DPDF* is easy to use without many parameters tuning. In addition, the performance of *DPDF* under different parameter settings is relatively stable.

### 6.2. The impact of the number of forest classifiers

In this section, we discuss the effect of the number of forests in each layer. We change the value of *M*, which represents the number of forests in each layer, from 3 to 5.

The results of our experiment are shown in Table 4. We can see that $M = 4$ obtains the best performance in 11 software projects from

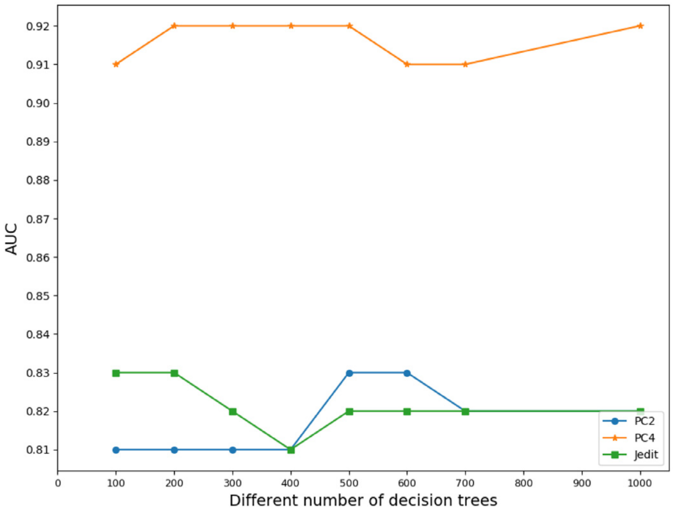

**Fig. 6.** Performance of DPDF under different parameter settings.

Table 2. Although $M = 4$ and $M = 5$ achieves similar AUC results, but when $M = 5$, we need more computing resources. It is possible that we can combine more than 5 forests in each layer, which may obtain better prediction performance. However, in fact, the more forests in each layer, the more computing resources are needed, which leads to a long-time running. So we choose 4 forests in each layer to build our model.

### 6.3. The impact of different forest classifiers

Besides the random forest, there are also other types of forest classifiers, which can be divided into two categories, i.e., forest classifiers based on bagging and boosting algorithms, respectively. In the bagging category, random forest and completely-random forest are the two most famous forest classifiers. Different from the random forest, completely-random tree is generated by randomly selecting a feature for split at each node of the tree, and growing tree until each leaf node contains only the same class of instances. In the boosting category, the most famous

**Table 4**
The AUC results of the number of forests in each layer.

| Dataset | Project | $M = 4$ | $M = 3$ | $M = 5$ |
|---|---|---|---|---|
| **NASA** | JW1 | **0.69** | 0.68 | 0.69 |
| | MC1 | 0.79 | 0.79 | **0.80** |
| | MC2 | **0.72** | 0.69 | 0.71 |
| | MW1 | 0.75 | 0.71 | **0.76** |
| | PC1 | **0.87** | 0.87 | 0.86 |
| | PC2 | 0.83 | 0.82 | **0.85** |
| | PC3 | **0.82** | 0.81 | 0.80 |
| | PC4 | **0.92** | 0.92 | 0.92 |
| | PC5 | 0.77 | 0.77 | **0.79** |
| **PROMISE** | Xalan v2.6 | **0.86** | 0.85 | 0.85 |
| | Ant v1.7 | **0.82** | 0.82 | 0.79 |
| | Camel v1.6 | **0.72** | 0.70 | 0.69 |
| | Jedit v4.0 | **0.82** | 0.81 | 0.82 |
| | Log4j v1.0 | 0.87 | 0.85 | **0.88** |
| | Lucene v2.4 | **0.79** | 0.76 | 0.73 |
| | Poi v3.0 | 0.88 | 0.86 | **0.89** |
| | Tomcat v6.0 | **0.89** | 0.89 | 0.87 |
| Average | | **0.81** | 0.80 | 0.81 |

**Table 5**
The AUC results of different forest classifiers.

| Dataset | Project | DPDF | $D^{XGBoost}$ | $D^{CompRF}$ |
|---|---|---|---|---|
| **NASA** | JW1 | **0.69** | 0.68 | 0.69 |
| | MC1 | **0.79** | 0.77 | 0.78 |
| | MC2 | **0.72** | 0.60 | 0.70 |
| | MW1 | **0.75** | 0.68 | 0.74 |
| | PC1 | 0.87 | 0.83 | **0.88** |
| | PC2 | **0.83** | 0.81 | 0.72 |
| | PC3 | 0.82 | 0.81 | **0.85** |
| | PC4 | 0.92 | **0.93** | 0.91 |
| | PC5 | 0.77 | 0.77 | **0.78** |
| **PROMISE** | Xalan v2.6 | **0.86** | 0.84 | 0.86 |
| | Ant v1.7 | 0.82 | **0.83** | 0.81 |
| | Camel v1.6 | **0.72** | 0.69 | 0.70 |
| | Jedit v4.0 | **0.82** | 0.80 | 0.79 |
| | Log4j v1.0 | **0.87** | 0.82 | 0.81 |
| | Lucene v2.4 | **0.79** | 0.79 | 0.75 |
| | Poi v3.0 | **0.88** | 0.86 | 0.85 |
| | Tomcat v6.0 | **0.89** | 0.82 | 0.89 |
| Average | | **0.81** | 0.78 | 0.80 |

forest classifier is eXtreme Gradient Boosting (XGBoost) [66]. XGBoost is a scalable tree boosting system, which is a highly effective and widely used tree boosting method. In this section, we also investigate the performance of our approach with different forest classifiers, and we refer to the DPDF with completely-random forest and XGBoost as $D^{CompRF}$ and $D^{XGBoost}$, respectively.

As Table 5 shows, our *DPDF* with random forest as the underlying classifier achieves the best AUC scores compared with other two forest classifiers. Thus, in practice, we recommend developers to use random forest as the underlying classifier.

## 6.4. Evaluation of other measures

In the task of defect prediction, many evaluation measures are used, and the most practical one is the accuracy measurement [8]. So we additionally report the accuracy of all the methods on 25 projects. The results can be seen in Table 6. We can see that our model is significantly better than other methods and has achieved the best results on 23 projects. In addition, we also report precision, recall, and F1 of all methods on 25 projects. Precision represents the rate of correctly predicted buggy instances among all instances predicted as buggy; recall measures the rate of correctly predicted buggy instances among all actual buggy instances; and F1 represents the harmonic mean of precision and recall [20]. The results are shown in Table 7, Tables 8 and Table 9. From these results, we notice that the average F1 and precision values of our approach are improved, with the recall value lower than the NB classifier.

## 6.5. Other layer-by-layer structure model

Inspired by logistic model trees [67], we try to use our cascade strategy on the logistic regression model. This model is named deep logistic regression (DLR). In each layer, we set four logistic regression classifiers. Table 10 shows the AUC results of this model on 25 datasets. From Table 10, we can easily see that DLR has better performance than LR.

**Table 6**
The Accuracy results of all classifiers.

| Dataset | Project | DPDF | gcForest | DBN | NB | LR | RF | SVM |
|---|---|---|---|---|---|---|---|---|
| **NASA** | JW1 | 0.798 | **0.800** | 0.785 | 0.783 | 0.788 | 0.778 | 0.790 |
| | MC1 | **0.983** | 0.977 | 0.977 | 0.678 | 0.973 | 0.977 | 0.977 |
| | MC2 | **0.746** | 0.660 | 0.648 | 0.712 | 0.664 | 0.688 | 0.680 |
| | MW1 | **0.910** | 0.885 | 0.853 | 0.609 | 0.814 | 0.867 | 0.893 |
| | PC1 | **0.926** | 0.913 | 0.913 | 0.869 | 0.894 | 0.909 | 0.913 |
| | PC2 | **0.982** | 0.975 | 0.937 | 0.765 | 0.949 | 0.975 | 0.979 |
| | PC3 | **0.900** | 0.876 | 0.876 | 0.506 | 0.879 | 0.866 | 0.875 |
| | PC4 | 0.889 | 0.860 | 0.862 | 0.791 | **0.894** | 0.878 | 0.887 |
| | PC5 | **0.776** | 0.752 | 0.723 | 0.745 | 0.739 | 0.756 | 0.747 |
| **PROMISE** | Xalan v2.6 | **0.762** | 0.738 | 0.536 | 0.711 | 0.738 | 0.743 | 0.731 |
| | Ant v1.7 | **0.832** | 0.809 | 0.777 | 0.805 | 0.820 | 0.815 | 0.813 |
| | Camel v1.6 | **0.808** | 0.802 | 0.805 | 0.790 | 0.801 | 0.800 | 0.808 |
| | Jedit v4.0 | **0.807** | 0.783 | 0.755 | 0.784 | 0.755 | 0.780 | 0.768 |
| | Log4j v1.0 | **0.792** | 0.780 | 0.748 | 0.755 | 0.733 | 0.759 | 0.756 |
| | Lucene v2.4 | **0.715** | 0.680 | 0.597 | 0.612 | 0.703 | 0.677 | 0.676 |
| | Poi v3.0 | **0.790** | 0.790 | 0.636 | 0.548 | 0.762 | 0.782 | 0.760 |
| | Tomcat v6.0 | **0.913** | 0.912 | 0.910 | 0.857 | 0.906 | 0.909 | 0.911 |
| **AEEEM** | LC | **0.930** | 0.922 | 0.907 | 0.851 | 0.841 | 0.917 | 0.907 |
| | JDT | **0.850** | 0.820 | 0.793 | 0.839 | 0.840 | 0.840 | 0.849 |
| | PDE | **0.870** | 0.867 | 0.86 | 0.835 | 0.856 | 0.865 | 0.861 |
| | EQ | **0.780** | 0.743 | 0.602 | 0.725 | 0.691 | 0.758 | 0.759 |
| | ML | **0.870** | 0.858 | 0.868 | 0.831 | 0.861 | 0.865 | 0.868 |
| **Relink** | Apache | **0.726** | 0.656 | 0.448 | 0.624 | 0.644 | 0.691 | 0.660 |
| | Safe | **0.750** | 0.627 | 0.607 | 0.696 | 0.732 | 0.705 | 0.571 |
| | Zxing | **0.704** | 0.678 | 0.704 | 0.657 | 0.679 | 0.679 | 0.692 |
| Average | | **0.832** | 0.807 | 0.765 | 0.735 | 0.798 | 0.811 | 0.805 |

**Table 7**
The F1 results of all classifiers.

| Dataset | Project | DPDF | gcForest | DBN | NB | LR | RF | SVM |
|---|---|---|---|---|---|---|---|---|
| **NASA** | JW1 | 0.23 | 0.15 | 0.18 | 0.28 | 0.18 | 0.24 | 0.13 |
| | MC1 | 0.04 | 0 | 0.04 | 0.09 | 0.08 | 0.18 | 0 |
| | MC2 | 0.48 | 0.44 | 0.41 | 0.45 | 0.56 | 0.46 | 0.35 |
| | MW1 | **0.51** | 0.13 | 0.22 | 0.26 | 0.18 | 0.06 | 0 |
| | PC1 | 0.17 | 0.07 | 0.27 | 0.35 | 0.3 | 0.22 | 0.09 |
| | PC2 | **0.83** | 0 | 0 | 0.07 | 0.12 | 0 | 0 |
| | PC3 | 0.11 | 0 | 0.22 | 0.31 | 0.35 | 0.17 | 0.01 |
| | PC4 | 0.33 | 0.23 | 0.51 | 0.37 | 0.53 | 0.35 | 0.34 |
| | PC5 | **0.46** | 0.37 | 0.37 | 0.33 | 0.35 | 0.44 | 0.26 |
| **PROMISE** | Xalan v2.6 | **0.72** | 0.7 | 0.68 | 0.6 | 0.69 | 0.7 | 0.68 |
| | Ant v1.7 | **0.55** | 0.46 | 0.48 | 0.55 | 0.48 | 0.49 | 0.43 |
| | Camel v1.6 | 0.19 | 0.16 | 0.19 | 0.33 | 0.21 | 0.27 | 0.07 |
| | Jedit v4.0 | **0.46** | 0.38 | 0.46 | 0.46 | 0.4 | 0.41 | 0.26 |
| | Log4j v1.0 | 0.48 | 0.39 | 0.63 | 0.48 | 0.45 | 0.41 | 0.21 |
| | Lucene v2.4 | **0.75** | 0.73 | 0.75 | 0.57 | 0.74 | 0.73 | 0.73 |
| | Poi v3.0 | **0.83** | 0.83 | 0.82 | 0.49 | 0.81 | 0.82 | 0.81 |
| | Tomcat v6.0 | 0.21 | 0.07 | 0 | 0.35 | 0.28 | 0.2 | 0.02 |
| **AEEEM** | LC | 0.37 | 0.36 | 0 | 0.39 | 0.3 | 0.34 | 0.03 |
| | JDT | 0.56 | 0.49 | 0.55 | 0.57 | 0.57 | 0.51 | 0.52 |
| | PDE | 0.31 | 0.16 | 0.28 | 0.4 | 0.35 | 0.27 | 0.05 |
| | EQ | **0.75** | 0.67 | 0.59 | 0.59 | 0.61 | 0.69 | 0.68 |
| | ML | 0.26 | 0.13 | 0.24 | 0.36 | 0.31 | 0.26 | 0.02 |
| **Relink** | Apache | **0.73** | 0.71 | 0.65 | 0.58 | 0.64 | 0.68 | 0.67 |
| | Safe | 0.56 | 0.42 | 0.64 | 0.56 | 0.56 | 0.54 | 0.33 |
| | Zxing | 0.29 | 0.25 | 0.17 | 0.27 | 0.31 | 0.39 | 0.11 |
| Average | | **0.45** | 0.33 | 0.37 | 0.40 | 0.41 | 0.39 | 0.27 |

**Table 8**
The Precision results of all classifiers.

| Dataset | Project | DPDF | gcForest | DBN | NB | LR | RF | SVM |
|---|---|---|---|---|---|---|---|---|
| **NASA** | JW1 | 0.49 | 0 | 0.49 | 0.53 | 0.49 | 0.41 | 0.91 |
| | MC1 | 0.29 | 0 | 0.03 | 0.05 | 0.12 | 0.49 | 0 |
| | MC2 | 0.60 | 0.50 | 0.33 | 0.57 | 0.51 | 0.58 | 0.65 |
| | MW1 | **0.63** | 0.14 | 0.31 | 0.16 | 0.17 | 0.06 | 0 |
| | PC1 | 0.25 | 0.27 | 0.86 | 0.32 | 0.30 | 0.35 | 0.45 |
| | PC2 | **0.98** | 0 | 0 | 0.04 | 0.10 | 0.01 | 0 |
| | PC3 | 0.26 | 0 | 0.51 | 0.20 | 0.53 | 0.29 | 0.01 |
| | PC4 | 0.77 | 0.41 | 0.70 | 0.30 | 0.64 | 0.58 | 0.89 |
| | PC5 | 0.61 | 0.59 | 0.58 | 0.58 | 0.53 | 0.57 | 0.69 |
| **PROMISE** | Xalan v2.6 | 0.76 | 0.74 | 0.77 | 0.83 | 0.75 | 0.76 | 0.77 |
| | Ant v1.7 | 0.64 | 0.61 | 0.56 | 0.57 | 0.65 | 0.61 | 0.7 |
| | Camel v1.6 | 0.46 | 0.29 | 0.53 | 0.42 | 0.84 | 0.47 | 0.28 |
| | Jedit v4.0 | **0.64** | 0.52 | 0.54 | 0.56 | 0.51 | 0.54 | 0.41 |
| | Log4j v1.0 | 0.65 | 0.50 | 0.88 | 0.45 | 0.46 | 0.52 | 0.27 |
| | Lucene v2.4 | 0.69 | 0.70 | 0.70 | 0.81 | 0.77 | 0.74 | 0.72 |
| | Poi v3.0 | **0.84** | 0.84 | 0.79 | 0.82 | 0.81 | 0.84 | 0.82 |
| | Tomcat v6.0 | **0.84** | 0.33 | 0 | 0.29 | 0.47 | 0.35 | 0.50 |
| **AEEEM** | LC | **0.81** | 0.72 | 0 | 0.31 | 0.26 | 0.65 | 0.05 |
| | JDT | 0.72 | 0.61 | 0.64 | 0.65 | 0.65 | 0.67 | 0.74 |
| | PDE | 0.59 | 0.29 | 0.79 | 0.4 | 0.47 | 0.54 | 0.54 |
| | EQ | 0.70 | 0.63 | 0.64 | 0.72 | 0.62 | 0.69 | 0.72 |
| | ML | **0.47** | 0.23 | 0.33 | 0.36 | 0.44 | 0.41 | 0.25 |
| **Relink** | Apache | **0.70** | 0.69 | 0.68 | 0.55 | 0.65 | 0.68 | 0.66 |
| | Safe | 0.57 | 0.44 | 0.83 | 0.58 | 0.58 | 0.67 | 0.42 |
| | Zxing | **0.47** | 0.33 | 0.47 | 0.41 | 0.41 | 0.42 | 0.26 |
| Average | | **0.62** | 0.42 | 0.52 | 0.46 | 0.51 | 0.52 | 0.47 |

### 6.6. Threats to validity

**Threats to construct validity** relate to the performance measures used in our study. In our work, our findings are based on one evaluation measure (AUC), and other evaluation measures may yield different results. However, unlike other measures (*precision, recall, accuracy and f-measure*) which need a cut-off value on the predicted probability of defect proneness, AUC is independent of a cut-off value. What's more, AUC is a widely used measure to evaluate the defect prediction techniques.

**Threats to internal validity** are mainly concerned with the uncontrolled internal factors that might have influence on the experimental results. The main internal threat is the potential faults during our method implementation. To reduce this threat, we used four classifiers which are obtained from scikit-learn libraries, one deep belief networks and one latest deep forest model called gcForest. For our proposed method *DPDF*, we designed various test cases to test the developed prototype and we prefer to use mature third-party libraries, such as packages from python. As discussed in Section 6, *DPDF* adopted the default values obtained in the official scikit-learn documentation for the parameters to perform the defect prediction. Other settings of the parameters may produce different results. We will study how to set the values for these parameters in the most optimized way that can produce better results in our future work.

**Table 9**
The Recall results of all classifiers.

| Dataset | Project | DPDF | gcForest | DBN | NB | LR | RF | SVM |
|---|---|---|---|---|---|---|---|---|
| **NASA** | JW1 | 0.15 | 0.07 | 0.11 | 0.19 | 0.11 | 0.17 | 0.07 |
| | MC1 | 0.02 | 0 | 0.06 | 0.5 | 0.06 | 0.11 | 0 |
| | MC2 | 0.4 | 0.39 | 0.54 | 0.37 | 0.62 | 0.38 | 0.24 |
| | MW1 | 0.43 | 0.12 | 0.17 | 0.64 | 0.19 | 0.06 | 0.04 |
| | PC1 | 0.13 | 0.02 | 0.16 | 0.39 | 0.3 | 0.16 | 0.05 |
| | PC2 | **0.72** | 0 | 0 | 0.29 | 0.14 | 0 | 0 |
| | PC3 | 0.07 | 0.01 | 0.14 | 0.74 | 0.26 | 0.12 | 0.01 |
| | PC4 | 0.21 | 0.16 | 0.4 | 0.49 | 0.45 | 0.25 | 0.21 |
| | PC5 | **0.37** | 0.27 | 0.22 | 0.23 | 0.26 | 0.36 | 0.16 |
| **PROMISE** | Xalan v2.6 | **0.68** | 0.66 | 0.61 | 0.47 | 0.64 | 0.65 | 0.61 |
| | Ant v1.7 | 0.48 | 0.37 | 0.42 | 0.53 | 0.38 | 0.41 | 0.31 |
| | Camel v1.6 | 0.12 | 0.11 | 0.1 | 0.27 | 0.12 | 0.19 | 0.04 |
| | Jedit v4.0 | 0.36 | 0.3 | 0.4 | 0.39 | 0.33 | 0.33 | 0.19 |
| | Log4j v1.0 | 0.38 | 0.32 | 0.49 | 0.51 | 0.44 | 0.34 | 0.17 |
| | Lucene v2.4 | **0.82** | 0.76 | 0.81 | 0.44 | 0.71 | 0.72 | 0.74 |
| | Poi v3.0 | 0.82 | 0.82 | 0.85 | 0.35 | 0.81 | 0.8 | 0.8 |
| | Tomcat v6.0 | 0.12 | 0.03 | 0 | 0.45 | 0.2 | 0.14 | 0.01 |
| **AEEEM** | LC | 0.24 | 0.24 | 0 | 0.52 | 0.35 | 0.23 | 0.02 |
| | JDT | 0.46 | 0.41 | 0.48 | 0.51 | 0.51 | 0.41 | 0.4 |
| | PDE | 0.21 | 0.11 | 0.17 | 0.4 | 0.28 | 0.18 | 0.02 |
| | EQ | **0.81** | 0.72 | 0.55 | 0.5 | 0.6 | 0.69 | 0.64 |
| | ML | 0.18 | 0.09 | 0.19 | 0.36 | 0.24 | 0.19 | 0.01 |
| **Relink** | Apache | **0.76** | 0.73 | 0.62 | 0.61 | 0.63 | 0.68 | 0.68 |
| | Safe | **0.55** | 0.4 | 0.52 | 0.54 | 0.54 | 0.45 | 0.27 |
| | Zxing | 0.21 | 0.2 | 0.09 | 0.2 | 0.25 | 0.36 | 0.07 |
| Average | | 0.39 | 0.29 | 0.32 | 0.44 | 0.38 | 0.34 | 0.23 |

**Table 10**
The AUC results of other layer-by-layer structure.

| Dataset | Project | DLR ($M = 4$) | LR | DPDF |
|---|---|---|---|---|
| **NASA** | JW1 | 0.68 | 0.67 | **0.69** |
| | MC1 | 0.41 | 0.63 | **0.79** |
| | MC2 | **0.79** | 0.66 | 0.72 |
| | MW1 | 0.59 | 0.48 | **0.75** |
| | PC1 | 0.82 | 0.76 | **0.87** |
| | PC2 | 0.43 | 0.55 | **0.83** |
| | PC3 | 0.66 | 0.81 | **0.82** |
| | PC4 | 0.91 | 0.90 | **0.92** |
| | PC5 | 0.61 | 0.72 | **0.77** |
| **PROMISE** | Xalan v2.6 | 0.81 | 0.80 | **0.86** |
| | Ant v1.7 | 0.78 | 0.79 | **0.82** |
| | Camel v1.6 | 0.68 | 0.67 | **0.72** |
| | Jedit v4.0 | 0.79 | 0.72 | **0.82** |
| | Log4j v1.0 | **0.90** | 0.72 | 0.87 |
| | Lucene v2.4 | 0.75 | 0.76 | **0.79** |
| | Poi v3.0 | 0.82 | 0.79 | **0.88** |
| | Tomcat v6.0 | 0.85 | 0.78 | **0.89** |
| **AEEEM** | LC | 0.71 | 0.55 | **0.82** |
| | JDT | 0.82 | 0.78 | **0.86** |
| | PDE | 0.69 | 0.72 | **0.77** |
| | EQ | 0.84 | 0.66 | **0.85** |
| | ML | 0.73 | 0.70 | **0.82** |
| **Relink** | Apache | 0.74 | 0.70 | **0.75** |
| | Safe | 0.70 | 0.67 | **0.73** |
| | Zxing | 0.67 | 0.57 | **0.70** |
| Average | | 0.73 | 0.70 | **0.80** |

**Threats to external validity** relate to the possibility to generalize our results. We performed our study on 25 projects, which cannot indicate all kinds of software. However, these datasets are often used in previous defect prediction studies [21,36,37]. In addition, our study only considers the defect metrics in these datasets for the classifiers. Using different sets of defect metrics may generate different results.

## 7. Related work

### *7.1. Machine learning in software defect prediction*

Software defect prediction is essential to ensure the quality of software. In recent years, machine learning has been widely used in software quality assurance [9,10,12–15,68]. Many studies focused on the applications of machine learning in defect prediction [8,24,69–71], which show that machine learning models are built for two different defect prediction tasks: within-project defect prediction and cross-project defect prediction. In our work, we focus on the within-project defect prediction task.

For within-project defect prediction, using machine learning algorithms has become the main stream including supervised learning and unsupervised learning. Among these work, improving the traditional supervised machine learning algorithms seems to be a feasible way. Rong et al. focused on improving the SVM model to obtain better prediction effect [49]. In their work, a CBA-SVM software defect prediction model was proposed, which takes advantage of the non-linear computing ability of SVM model and optimization capacity of bat algorithm with centroid strategy. Rana et al. evaluated how information about defect inflow distribution from historical projects is applied for modeling the prior beliefs/experience in Bayesian analysis which is useful for software defect prediction at an early stage during the software project life cycle [72]. Okutan et al. used Bayesian networks to determine the probabilistic influential relationships among software metrics and defect proneness [50]. Arar et al. proposed a Feature Dependent Naive Bayes (FDNB) classification method, which shows that their approach is better than the standard Naive Bayes approach [51]. Agarwal et al. used the Bayesian hierarchical modeling that performs closely with manual inspection and predicts surface defects with relatively good accuracy [73].

In addition, unsupervised learning algorithms also attract a lot of studies in unlabeled datasets. Yang et al. proposed a new approach for predicting defect proneness on unlabeled datasets [74]. Maruf et al. presented a new defect clustering method using *k*-means++ for web page source code and they showed that half of the defects can be detected in the web pages [75]. Yang et al. used the commonly change metrics to build simple unsupervised models, which shows that many simple unsupervised models perform better than the state-of-the-art supervised models in effort-aware Just-In-Time defect prediction [76]. Nam et al. proposed CLA and CLAMI, that show the potential for defect prediction on unlabeled datasets in an automated manner without the need for manual effort [42].

In our work, we choose supervised learning algorithms because existing studies show that the performance of supervised learning classifiers is better than unsupervised learning classifiers in most projects [34,77,78]. What's more, we use the cascade strategy on traditional supervised learning which can greatly improve the performance of the previous classifiers.

### 7.2. Feature selection in software defect prediction

Effective feature selection can improve the performance of classifiers. Xu et al. proposed a novel feature selection framework, MICHAC, based on the Maximal Information Coefficient with Hierarchical Agglomerative Clustering, which is shown to be effective in selecting features in defect prediction [79]. Liu et al. proposed a new feature selection framework using FEature Clustering And feature Ranking (FECAR) [80]. Wang et al. leveraged the Deep Belief Network (DBN) to automatically learn semantic features from token vectors extracted from programs' Abstract Syntax Trees, which can improve the performance of classifier in both within-project defect prediction and cross-project defect prediction [81]. Laradji et al. combined selected ensemble learning models with efficient feature selection to address software defect data including redundancy, correlation, feature irrelevance and missing samples, and mitigate their effects on the defect classification performance which shows only few features contribute to high area under the receiver-operating curve [82]. Xu et al. investigated the impact of 32 feature selection methods on the defect prediction performance over two versions of the NASA datasets and one open source AEEEM dataset, which shows that the effectiveness of these feature selection methods on defect prediction performance varies significantly over all the datasets [83].

In this paper, different from the work based on Abstract Syntax Trees, i.e., Wang et al. [81] and Li et al. [25], our work mainly used other types of features such as process features, and complexity features. Moreover, the setting of Wang et al.'s work is different from ours, their approach worked on the cross-version defect prediction (i.e., using the defective data from previous version to predict defective classes in the current version), while our approach worked in the within-version defect prediction. In the future, we plan to combine the two approaches together to further improve the performance of defect prediction.

## 8. Conclusion and future work

Machine learning is widely used for software defect prediction. In this paper, we propose a new defect prediction model (*DPDF*) based on deep forest, which is treated as an alternative to Deep Neural Networks. The main idea of our *DPDF* is to use the cascade strategy on traditional machine learning algorithm (random forest) to help select useful defect features and representation learning based on the layer-by-layer structure. Our experiments on four data sets show that *DPDF* can improve the performance over the state-of-the-art machine learning classifiers, moreover, *DPDF* with the cascade strategy can effectively improve the performance of defect prediction over the basic structure of *DPDF* (random forest).

In the future, we will explore the potential of our *DPDF* on cross project defect prediction on more projects. In addition, during the process of defect prediction based on all the input features, some features may not be related to the faults. We will combine other approaches to address the issue of feature subset selection for further improving our approach. Finally, we would like to explore the potential of the SMOTUNED method [84] proposed by Agrawal et al. to further address the class-imbalance problem in our defect prediction or provide a comprehensive study to compare with the SMOTE/SMOTUNED method on the imbalanced datasets.

## Acknowledgments

This work is supported partially by Natural Science Foundation of China under Grant No. 61872312, No. 61402396, No. 61472344 and No. 61611540347, partially by the Open Funds of State Key Laboratory for Novel Software Technology of Nanjing University under Grant no. KFKT2018B12, partially by the Jiangsu "333" Project, partially by the the Six Talent Peaks Project in Jiangsu Province under Grant No. RJFW-053, and partially by the Jiangsu Overseas Visiting Scholar program.

## References

[1] T. Hall, S. Beecham, D. Bowes, D. Gray, S. Counsell, A systematic literature review on fault prediction performance in software engineering, IEEE Trans. Softw. Eng. 38 (6) (2012) 1276–1304.
[2] X. Sun, X. Peng, K. Zhang, Y. Liu, Y. Cai, How security bugs are fixed and what can be improved: an empirical study with Mozilla, Sci. China Inf. Sci. 62 (1) (2018) 19102, doi:10.1007/s11432-017-9459-5.
[3] X. Sun, H. Yang, X. Xia, B. Li, Enhancing developer recommendation with supplementary information via mining historical commits, J. Syst. Softw. 134 (2017) 355–368, doi:10.1016/j.jss.2017.09.021.
[4] G. Tassey, The economic impacts of inadequate infrastructure for software testing (2002).
[5] L. Wang, X. Sun, J. Wang, Y. Duan, B. Li, Construct bug knowledge graph for bug resolution: poster, in: Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20–28, 2017 - Companion Volume, 2017, pp. 189–191, doi:10.1109/ICSE-C.2017.102.
[6] X. Sun, W. Zhou, B. Li, Z. Ni, J. Lu, Bug localization for version issues with defect patterns, IEEE Access 7 (2019) 18811–18820, doi:10.1109/ACCESS.2019.2894976.
[7] X. Sun, X. Peng, B. Li, B. Li, W. Wen, IPSETFUL: an iterative process of selecting test cases for effective fault localization by exploring concept lattice of program spectra, Front. Comput. Sci. 10 (5) (2016) 812–831, doi:10.1007/s11704-016-5226-y.
[8] M.J. Shepperd, D. Bowes, T. Hall, Researcher bias: the use of machine learning in software defect prediction, IEEE Trans. Softw. Eng. 40 (6) (2014) 603–616.
[9] X. Jing, F. Wu, X. Dong, F. Qi, B. Xu, Heterogeneous cross-company defect prediction by unified metric representation and CCA-based transfer learning, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, 2015, pp. 496–507.
[10] R. Malhotra, An empirical framework for defect prediction using machine learning techniques with android software, Appl. Soft Comput. 49 (2016) 1034–1050.
[11] X. Sun, T. Zhou, G. Li, J. Hu, H. Yang, B. Li, An empirical study on real bugs for machine learning programs, in: 24th Asia-Pacific Software Engineering Conference, APSEC 2017, Nanjing, China, December 4–8, 2017, 2017, pp. 348–357, doi:10.1109/APSEC.2017.41.
[12] H. Lu, E. Kocaguneli, B. Cukic, Defect prediction between software versions with active learning and dimensionality reduction, in: ISSRE '14 Proceedings of the 2014 IEEE 25th International Symposium on Software Reliability Engineering, 2014, pp. 312–322.
[13] T. Wang, Z. Zhang, X. Jing, L. Zhang, Multiple kernel ensemble learning for software defect prediction, Autom. Softw. Eng. 23 (4) (2016) 569–590.
[14] Z.-W. Zhang, X.-Y. Jing, T.-J. Wang, Label propagation based semi-supervised learning for software defect prediction, Autom. Softw. Eng. 24 (1) (2017) 47–69.
[15] Z. Li, X.-Y. Jing, X. Zhu, H. Zhang, Heterogeneous defect prediction through multiple kernel learning and ensemble learning, in: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2017, pp. 91–102.
[16] P.M. Domingos, M.J. Pazzani, On the optimality of the simple Bayesian classifier under zero-one loss, Mach. Learn. 29 (1997) 103–130.
[17] D.R. Cox, Two further applications of a model for binary regression, Biometrika 45 (1958) 562–565.
[18] L. Breiman, Random forests, Mach. Learn. 45 (1) (2001) 5–32.
[19] N. Cristianini, J. Shawe-Taylor, An Introduction to Support Vector Machines: and Other Kernel-Based Learning Methods, Printed in the United Kingdom at the University Press, 2000.
[20] S. Lessmann, B. Baesens, C. Mues, S. Pietsch, Benchmarking classification models for software defect prediction: a proposed framework and novel findings, IEEE Trans. Softw. Eng. 34 (4) (2008) 485–496.
[21] B. Ghotra, S. Mcintosh, A.E. Hassan, Revisiting the impact of classification techniques on the performance of defect prediction models, in: International Conference on Software Engineering, 2015, pp. 789–800.
[22] A. Krizhevsky, I. Sutskever, G.E. Hinton, ImageNet classification with deep convolutional neural networks, in: International Conference on Neural Information Processing Systems, 2012, pp. 1097–1105.
[23] G. Hinton, L. Deng, D. Yu, G.E. Dahl, A.R. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T.N. Sainath, Deep neural networks for acoustic modeling in speech recognition: the shared views of four research groups, IEEE Signal Process. Mag. 29 (6) (2012) 82–97.
[24] X. Yang, D. Lo, X. Xia, Y. Zhang, J. Sun, Deep learning for just-in-time defect prediction, in: IEEE International Conference on Software Quality, Reliability and Security, 2015, pp. 17–26.
[25] J. Li, P. He, J. Zhu, M.R. Lyu, Software defect prediction via convolutional neural network, in: IEEE International Conference on Software Quality, Reliability and Security, 2017.
[26] Z.-H. Zhou, J. Feng, Deep forest: towards an alternative to deep neural networks, in: Twenty-Sixth International Joint Conference on Artificial Intelligence, 2017, pp. 3553–3559.
[27] C. Cheadle, Y.S. Cho-Chung, K.G. Becker, M.P. Vawter, Application of *z*-score transformation to affymetrix data, Appl. Bioinf. 2 (4) (2003) 209–217.
[28] M.H. Halstead, Elements of software science, Elsevierence (1977).

[29] T.J. Mccabe, A complexity measure, IEEE Trans. Softw. Eng. SE-2 (4) (2006) 308–320.
[30] S.R. Chidamber, C.F. Kemerer, A Metrics Suite for Object Oriented Design, IEEE Press, 1994.
[31] N. Fenton, M. Neil, W. Marsh, P. Hearty, ukasz Radliski, P. Krause, On the effectiveness of early life cycle defect prediction with Bayesian nets, Empir. Softw. Eng. 13 (5) (2008) 499.
[32] D. Isa, R. Rajkumar, Pipeline defect prediction using support vector machines, Appl. Artif. Intell. 23 (8) (2009) 758–771.
[33] J. Wang, B. Shen, Y. Chen, Compressed c4.5 models for software defect prediction, in: International Conference on Quality Software, 2012, pp. 13–16.
[34] 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.
[35] Y. Bengio, A. Courville, P. Vincent, Representation Learning: A Review and New Perspectives, IEEE Computer Society, 2013.
[36] C. Tantithamthavorn, S. Mcintosh, A.E. Hassan, K. Matsumoto, Automated parameter optimization of classification techniques for defect prediction models, in: International Conference on Software Engineering, 2016, pp. 321–332.
[37] 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, 2014, pp. 414–423.
[38] M. Shepperd, Q. Song, Z. Sun, C. Mair, Data quality: some comments on the nasa software defect datasets, IEEE Trans. Softw. Eng. 39 (9) (2013) 1208–1215.
[39] M. Jureczko, L. Madeyski, Towards identifying software project clusters with regard to defect prediction, in: International Conference on Predictive MODELS in Software Engineering, 2010, p. 9.
[40] M. D'Ambros, M. Lanza, R. Robbes, An extensive comparison of bug prediction approaches, Mining Software Repositories, 2010.
[41] R. Wu, H. Zhang, S. Kim, S.C. Cheung, ReLink: recovering links between bugs and changes, in: ACM SIGSOFT Symposium & the European Conference on Foundations of Software Engineering, 2011.
[42] J. Nam, S.H. Kim, CLAMI: defect prediction on unlabeled datasets, in: The 30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015), Lincoln, Nebraska, USA, 2015, p. 1.
[43] X. Xia, D. Lo, S. McIntosh, E. Shihab, A.E. Hassan, Cross-project build co-change prediction, in: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2015, pp. 311–320.
[44] A.P. Bradley, The use of the area under the ROC curve in the evaluation of machine learning algorithms, Pattern Recognit. 30 (7) (1997) 1145–1159.
[45] C. Tantithamthavorn, A.E. Hassan, An experience report on defect modelling in practice: pitfalls and challenges, in: In Proceedings of the International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP18), 2018.
[46] F. Rahman, D. Posnett, P. Devanbu, Recalling the "imprecision" of cross-project defect prediction, in: SIGSOFT FSE, 2012, pp. 1–11.
[47] T. Fawcett, An introduction to ROC analysis, Pattern Recognit. Lett. 27 (8) (2006) 861–874.
[48] F. Gorunescu, Data Mining: Concepts, Models and Techniques, Springer, 2011.
[49] X. Rong, F. Li, Z. Cui, A model for software defect prediction using support vector machine based on cba, Int. J. Intell. Syst. Technol. Appl. 15 (1) (2016) 19–34.
[50] A. Okutan, O.T. Yldz, Software defect prediction using Bayesian networks, Empir. Softw. Eng. 19 (1) (2014) 154–181.
[51] mer Faruk Arar, K. Ayan, A feature dependent Naive Bayes approach and its application to the software defect prediction problem, Appl. Soft Comput. 59 (2017) 197–209.
[52] M. Pinzger, N. Nagappan, B. Murphy, Can developer-module networks predict failures? in: ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, pp. 2–12.
[53] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, A. Bener, Defect prediction from static code features: current results, limitations, new approaches, Autom. Softw. Eng. 17 (4) (2010) 375–407.
[54] B. Turhan, T. Menzies, A.B. Bener, J.D. Stefano, On the relative value of cross–company and within-company data for defect prediction, Empir. Softw. Eng. 14 (5) (2009) 540–578.
[55] J. Ar, Statistical comparisons of classifiers over multiple data sets, J. Mach. Learn. Res. 7 (1) (2006) 1–30.
[56] X. Xia, D. Lo, S.J. Pan, N. Nagappan, X. Wang, HYDRA: massively compositional model for cross-project defect prediction, IEEE Trans. Softw. Eng. 42 (10) (2016) 977–998.
[57] Z. He, F. Shu, Y. Yang, M. Li, Q. Wang, An investigation on the feasibility of cross-project defect prediction, Autom. Softw. Eng. 19 (2) (2012) 167–199.
[58] Y. Ma, G. Luo, X. Zeng, A. Chen, Transfer learning for cross-company software defect prediction, Inf. Softw. Technol. 54 (3) (2012) 248–256.
[59] X. Sun, B. Li, H.K.N. Leung, B. Li, Y. Li, MSR4SM: using topic models to effectively mining software repositories for software maintenance tasks, Inf. Softw. Technol. 66 (2015) 1–12, doi:10.1016/j.infsof.2015.05.003.
[60] X. Sun, H. Leung, B. Li, B. Li, Change impact analysis and changeability assessment for a change proposal: an empirical study 97349734, J. Syst. Softw. 96 (2014) 51–60, doi:10.1016/j.jss.2014.05.036.
[61] L. Chen, B. Fang, Z. Shang, Y. Tang, Negative samples reduction in cross-company software defects prediction, Inf. Softw. Technol. 62 (2015) 67–77.
[62] J. Romano, J.D. Kromrey, J. Coraggio, J. Skowronek, Appropriate statistics for ordinal level data: Should we really be using *t*-test and Cohen's *d* for evaluating group differences on the NSSE and other surveys?, 2006.
[63] X.Y. Jing, F. Wu, X. Dong, B. Xu, An improved SDA based defect prediction framework for both within-project and cross-project class-imbalance problems, IEEE Trans. Softw. Eng. 43 (4) (2017) 321–339.
[64] T. Lee, J. Nam, D. Han, S. Kim, H.P. In, Developer micro interaction metrics for software defect prediction, IEEE Trans. Softw. Eng. 42 (11) (2016) 1015–1035.
[65] F. Zhang, A. Mockus, I. Keivanloo, Y. Zou, Towards building a universal defect prediction model with rank transformed predictors, Empir. Softw. Eng. 21 (5) (2016) 2107–2145.
[66] T. Chen, C. Guestrin, XGBoost: a scalable tree boosting system, in: ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2016, pp. 785–794.
[67] N. Landwehr, M. Hall, E. Frank, Logistic model trees, Mach. Learn. 59 (1–2) (2005) 161–205.
[68] H. Wang, L. Wang, Q. Yu, Z. Zheng, A. Bouguettaya, M.R. Lyu, Online reliability prediction via motifs-based dynamic Bayesian networks for service-oriented systems, IEEE Trans. Softw. Eng. PP (99) (2017) 556–579.
[69] C. Tantithamthavorn, S. Mcintosh, A.E. Hassan, K. Matsumoto, An empirical comparison of model validation techniques for defect prediction models, IEEE Trans. Softw. Eng. 43 (1) (2017) 1–18.
[70] S. Herbold, Comments on ScottKnottESD in response to an empirical comparison of model validation techniques for defect prediction models, IEEE Trans. Softw. Eng. 43 (11) (2017) 1091–1094.
[71] X. Chen, Y. Zhao, Q. Wang, Z. Yuan, Multi: multi-objective effort-aware just-in-time software defect prediction, Inf Softw. Technol. 93 (2018) 1–13.
[72] R. Rana, M. Staron, C. Berger, J. Hansson, M. Nilsson, W. Meding, Analyzing defect inflow distribution and applying Bayesian inference method for software defect prediction in large software projects, J. Syst. Softw. 117 (2016) 229–244.
[73] K. Agarwal, R. Shivpuri, On line prediction of surface defects in hot bar rolling based on Bayesian hierarchical modeling, J. Intell. Manuf. 26 (4) (2015) 785–800.
[74] J. Yang, H. Qian, Defect prediction on unlabeled datasets by using unsupervised clustering, in: 2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2016, pp. 465–472.
[75] M.M. ztrk, U. Cavusoglu, A. Zengin, A novel defect prediction method for web pages using *k*-means++, Expert Syst. Appl. 42 (19) (2015) 6496–6506.
[76] 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, 2016, pp. 157–168.
[77] M. Yan, Y. Fang, D. Lo, X. Xia, X. Zhang, File-level defect prediction: unsupervised vs. supervised models, in: 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 2017, pp. 344–353.
[78] 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.
[79] Z. Xu, J. Xuan, J. Liu, X. Cui, MICHAC: defect prediction via feature selection based on maximal information coefficient with hierarchical agglomerative clustering, in: IEEE International Conference on Software Analysis, Evolution, and Reengineering, 2016, pp. 370–381.
[80] S. Liu, X. Chen, W. Liu, J. Chen, Q. Gu, D. Chen, FECAR: a feature selection framework for software defect prediction, in: Computer Software and Applications Conference, 2014, pp. 426–435.
[81] S. Wang, T. Liu, L. Tan, Automatically learning semantic features for defect prediction, in: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), 2016, pp. 297–308.
[82] I.H. Laradji, M. Alshayeb, L. Ghouti, Software defect prediction using ensemble learning on selected features, Inf. Softw. Technol. 58 (2015) 388–402.
[83] Z. Xu, J. Liu, Z. Yang, G. An, X. Jia, The impact of feature selection on defect prediction performance: an empirical comparison, in: 2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE), 2016, pp. 309–320.
[84] A. Agrawal, T. Menzies, Is "better data" better than "better data miners"?: on the benefits of tuning SMOTE for defect prediction, in: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27, - June 03, 2018, 2018, pp. 1050–1061, doi:10.1145/3180155.3180197.