Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

Automatic Feature Exploration and An Application in Defect Prediction

YU QIU[1]*, Yun Liu[2]*, Ao Liu[2], Jingwen Zhu[3], and Jing Xu[1]
[1]College of Artificial Intelligence, Nankai University, Tianjin 300350, China
[2]College of Computer Science, Nankai University, Tianjin 300350, China
[3]College of Software, Nankai University, Tianjin 300350, China
*Equal contribution.

Corresponding author: Jing Xu (e-mail: xujing@nankai.edu.cn).

This work was supported by Science and Technology Planning Project of Tianjin (17JCZDJC30700 and 18ZXZNGX00310), Fundamental Research Funds for the Central Universities of Nankai University (63191402).

ABSTRACT Many software engineering tasks heavily rely on hand-crafted software features, *e.g.*, defect prediction, vulnerability discovery, software requirements, code review, and malware detection. Previous solutions to these tasks usually directly use the hand-crafted features or feature selection techniques for classification or regression, which usually leads to suboptimal results due to their lack of powerful representations of the hand-crafted features. To address the above problem, in this paper, we adopt the effort-aware just-in-time software defect prediction (JIT-SDP), which is a typical hand-crafted-feature-based task, as an example, to exploit new possible solutions. We propose a new model, named *neural forest* (NF), which uses the deep neural network and decision forest to build a holistic system for the automatic exploration of powerful feature representations that are used for the following classification. NF first employs a deep neural network to learn new feature representations from hand-crafted features. Then, a decision forest is connected after the neural network to perform classification and in the meantime, to guide the learning of feature representation. NF mainly aims at solving the challenging problem of combining the two different worlds of neural networks and decision forests in an end-to-end manner. When compared with previous state-of-the-art defect predictors and five designed baselines on six well-known benchmarks for within- and cross-project defect prediction, NF achieves significantly better results. The proposed NF model is generic to the classification problems which rely on the hand-crafted features.

INDEX TERMS Feature exploration, hand-crafted features, defect prediction

I. INTRODUCTION

HAND-CRAFTED software features play a core role in many software engineering tasks, such as defect prediction, software vulnerability discovery, software requirements, software code review, and malware detection. Traditional models usually perform simple feature pre-processing, *e.g.*, feature selection and combination, and further adopt machine learning models, *e.g.*, support vector machines (SVMs) and random forest, to classify or regress for the targets. The problem is that it is difficult to fully leverage the latent correlations and interconnections among features using manually designed feature pre-processing techniques. In this paper, we select a typical task in software engineering, just-in-time software defect prediction (JIT-SDP), including within-project and cross-project prediction to exploit this problem.

Different from module-level (*e.g.*, package-level, file-level or class-level) defect prediction [1]–[6], JIT-SDP predicts defects at the change-level and identifies defect-inducing changes right after the code is committed [7]–[12]. JIT-SDP is more practical for large software systems. On the one hand, it can locate the defects at the fine-granularity level and help developers narrow down the code needed for inspection. On the other hand, it can reduce the cost of revising the code when we identify the developers who introduce the defects. Furthermore, effort-aware JIT-SDP takes into consideration the effort needed to inspect the code that is related to the defect-inducing changes. It is more important in practice due to the limited debugging resources and tight development schedules [9], [11], [13]. Hence we mainly focus on effort-aware JIT-SDP in this paper.

Kamei *et al.* [9] first introduced the effort consideration into JIT-SDP and used 14 hand-crafted features to build the

change risk prediction model. This change risk model can identify 35% of all defect-inducing changes when only 20% effort is spent. With a wide range of software factors as inputs (*e.g.*, features in [9]), many effort-aware JIT-SDP predictors [9], [14]–[20] manually remove some highly correlated features to deal with the risk of multicollinearity and then use machine learning models to classify a change to be buggy or clean. However, directly removing correlated features is sub-optimal because every feature has particular characteristics, even when correlated with other features. Motivated by this, some studies employ deep belief networks (DBNs) to extract feature representations from typical hand-crafted features [4], [21], [22]. However, DBNs are separately trained feature extractors, so the extracted feature representations may be not optimal for consequent classification, while what we expect here is an end-to-end trained system consisting of a feature extractor and a consequent classifier. End-to-end learning is important to obtain an optimal model as well accepted in the machine learning community [23]–[25].

Aiming at above problems, this paper presents a novel model, named *neural forest* (**NF**), to automatically learn proper representations from traditional features [9] by optimizing the feature extractor together with the consequent classifier. First, NF leverages the powerful representation capability of the deep neural network to extract high-dimensional feature representations from traditional features [9]. Then, a reformative decision forest is connected after the neural network to classify the corresponding changes to be buggy or clean using the generated comprehensive feature representations. We manage to bridge these two different worlds (*i.e.*, neural network and decision forest) into a holistic model and train the model using back-propagation [26], [27] through joint and global optimization. The end-to-end training can help the network learn proper feature representations for the decision forest (classifier), and the optimization of decision forest attempts to better classify the features generated by the neural network. Hence our model can maximize the potential information hidden in the hand-crafted features through the global optimization of the two sub-models.

The reason why we use decision forests [28] as the classifier is that decision forests have outstanding performance in dealing with high-dimensional data problems [29]. Hence decision forests are suitable to classify the high-dimensional feature representations generated by neural networks. The decision forest [28] integrates many decision trees using bootstrap aggregation (bagging). In typical decision trees, a decision node is binary, and the routing is fixed. In other words, when a sample reaches a decision node, it will deterministically go along the left or right subtree in the next step. In this paper, we consider a reformative decision tree which has a probabilistic routing [30]–[32]. Specifically, when a sample reaches a decision node, it will reach the left and right subtree with a probability. Furthermore, our NF model is also generic for other software engineering tasks discussed at the beginning.

We compare with four state-of-the-art models on six large-scale open source projects, *i.e.*, Bugzilla, Columba, JDT, Mozilla, Platform, and PostgreSQL, using five widely used metrics, *i.e.*, recall, precision, F1-score, AUC and P_{opt}. Moreover, we design five baselines which apply the feature selection techniques to identify the subsets of more representative features and then use random forests or support vector machine (SVM) to classify these selected features. We compare our model with these five baselines in terms of the five metrics above to further demonstrate the effectiveness of our model. Experimental results show that NF can significantly improve the performance in terms of all evaluation metrics on all datasets. [1]

The main contributions of this paper include:

- We design a deep neural network for learning high-dimensional feature representations from traditional features automatically, and a reformative decision forest is connected after the neural network to classify the learned features.
- With the derivation of optimization formulas, we manage to optimize the proposed NF model in an end-to-end manner, so that the learned features are distinctive enough.
- We use JIT-SDP to demonstrate the classification superiority of NF and show the ways of analyses for hand-crafted features.

II. RELATED WORK

In this section, we first introduce effort-aware JIT-SDP that we mainly focus on in this paper. Then, we introduce some feature selection methods used in the defect prediction models.

A. EFFORT-AWARE JIT-SDP

With the development of traditional defect prediction that aims to identify whether a module (*e.g.*, package, file, or class) is defect-prone [1]–[3], [33]–[41], Mockus *et al.* [7] proposed the change-level software defect prediction, which conducted defect prediction at the change level. Such fine-granularity defect prediction can narrow the scope of code inspection [37] and thus has more applications in practice. Mockus *et al.* [7] predicted whether a change was defect-inducing based on a number of change characteristics, such as the lines of added/deleted code, the diffusion of the change, the experience of developers, *etc*. Later, a large amount of literature on change-level software defect prediction has appeared [8], [9], [13], [42], [43].

Kamei *et al.* [9] first named the change-level defect prediction as just-in-time software defect prediction (JIT-SDP). Moreover, the authors pointed out that the effort should be considered for JIT-SDP, *i.e.*, effort-aware JIT-SDP, and used the number of the lines of code (LOC) added and deleted in a change to measure the needed effort when inspecting

[1]The code and data of this paper will be available at https://github.com/yun-liu/NF-DefectPrediction.

this change, which is widely accepted in [9], [20], [44]–[46]. Since it is usually unpractical for developers to check all the code predicted as buggy due to the limited debugging resources and tight development schedules, we should distinguish as many defects as possible with a certain effort.

Motivated by [9], Yang *et al.* [21] leveraged a deep learning method to improve the performance of JIT-SDP. The authors first employed DBN to extract a set of expressive features from hand-crafted features. A classifier was then trained to classify the buggy or clean changes based on these expressive features. Yang *et al.* [16] built many unsupervised models to conduct effort-aware JIT-SDP. The authors compared these unsupervised methods with the supervised methods and concluded that simple unsupervised models had outstanding performance than supervised models. However, Fu *et al.* [18] rejected this conclusion [16] after a thorough analysis with experimental results. The authors found that the unsupervised models in [16] performed worse than supervised models in some settings. Moreover, the authors built a new classifier, OneWay, which was a supervised predictor built on the implication of simple unsupervised predictors from [16]. This OneWay learner achieves the state-of-the-art performance. Similarly, Huang *et al.* [17] made a holistic comparison with supervised and unsupervised models. The authors pointed out that LT, a best unsupervised model built by Yang *et al.* [16], finds more defective changes and many highly ranked changes are false alarms. According to this finding, the authors proposed an improved supervised model CBS and significantly improved the precision and F1-score.

In recent years, many other supervised and unsupervised models (*e.g.*, CCUM [14], TLEL [19], and MULTI [20]) have been proposed to push forward the state of the arts. Fukushima *et al.* [47] found given a project with a few historical changes, we could use the changes from other projects as the training data to build the prediction model, as followed by other research [48], [49]. More recently, some study [50], [51] focuses on the impact of classification techniques, and some other study [52]–[57] attempts to handle the problem of imbalanced data in effort-aware JIT-SDP. Tourani *et al.* [10] built logistic regression models to study the impact of the characteristics of issue and review discussions on the defect-proneness of a patch.

B. FEATURE SELECTION METHODS

Feature selection [58]–[62] is a family of data preprocessing techniques which automatically select a subset of more representative features to replace the original set. Classic feature selection techniques, such as information gain [61], [63], chi-square statistics [58] and distance correlation [64], [65] are employed to investigate the defect features. In general, a standard feature selection algorithm usually ranks features according to their relevance scores. The larger a score is, the better the attribute is to distinguish between potential classes. Then, the top-ranked features are selected as a subset of representative features for defect prediction.

Previous defect prediction models usually use feature selection techniques to remove correlated features for performance improvement. Kamei *et al.* [9] found that the features of NF and ND, REXP and EXP, and LA and LD were highly correlated, and thus excluded ND, REXP and LA from the model. Many following works [15]–[18], [47] perform similar data preparation to deal with the risk of multicollinearity. For example, Yang *et al.* [16] excluded the features of LA and LD from the candidate features and only used the other 12 features in Table 1.

For traditional module-level defect prediction, many studies use feature selection to improve prediction performance [66], [67]. Laradji *et al.* [68] carefully combined ensemble learning with efficient feature selection to address the issues of correlation, feature irrelevance, and so on. Jiarpakdee *et al.* [69] investigated the impact of correlated metrics and found that removing correlated metrics produced a more accurate and reliable interpretation of defect models while resulting in negligible effects on the performance and stability of the models. Since correlation analyses often involve the manual selection of metrics, Jiarpakdee *et al.* [70] proposed an automatic metric selection approach based on correlation analyses.

However, these methods rely on manually designed feature selection strategies or simply apply previous feature selection techniques, so it is difficult for them to take full advantage of the given hand-crafted features. In this paper, we propose a neural forest model which simultaneously learns the feature representations and classifier. The holistic training of the feature extractor and the classifier can automatically exploit correlations and interconnections among given features and thus leads to better prediction performance and reliable interpretation.

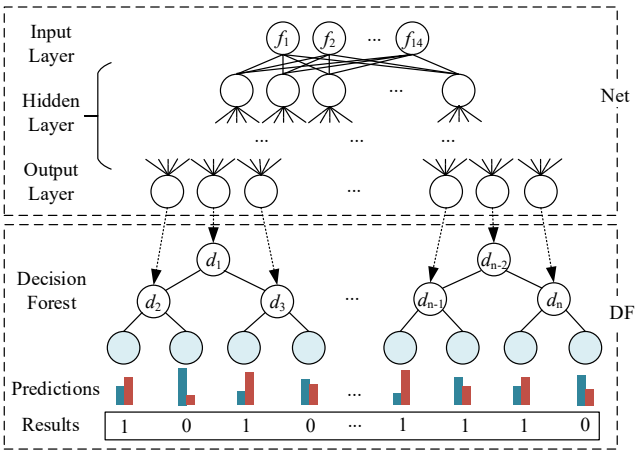


FIGURE 1. Illustration of our NF model. The **top part** (*i.e.*, Net) is the neural network and the **bottom part** (*i.e.*, DF) is the decision forest. Our designed fully connected neural network contains three parts: the input layer, hidden layers, and the output layer that is used for features representations learning. Our proposed probabilistic decision forest can be viewed as a classifier. The neurons ($\{f_1, \cdots, f_{14}\}$) in the input layer of neural network represent the 14 traditional features. The neurons in the output layer are used as the split nodes of decision trees in the probabilistic decision forest. The leaf nodes (*i.e.*, in light blue) of the forest contain corresponding probability distributions (*i.e.*, histograms) of the inputs being buggy or clean.

TABLE 1. Summary of hand-crafted change features (Pur. = Purpose).

Dim.	Feature	Description	Rationale
Diffusion	NS	Number of modified subsystems [7]	Changes modifying more subsystems are more likely to be defect-inducing.
	ND	Number of modified directories [7]	Changes which modify more directories are more likely to be defect-inducing.
	NF	Number of modified files [33]	Changes touching more files are more likely to be defect-inducing.
	Entropy	Distribution of modified code across each file [34], [71]	Changes with high entropy are more likely to be defect-inducing, because a developer will have to recall and track large numbers of scattered changes across each file.
Size	LA	Lines of added code [72]	The more lines of code added,the more likely a defect is introduced.
	LD	Lines of deleted code [72]	The more lines of code deleted,the higher the chance of a defect.
	LT	Lines of code in a file before the change [73]	The larger a file, the more likely a change might introduce a defect.
Pur.	FIX	Whether the change is used to fix a defect [74], [75]	Fixing a defect means that an error was made in an earlier implementation, therefore it may indicate an area where errors are more likely.
History	NDEV	Number of developers that changed the modified files [76]	The large the NDEV, the more likely a defect is introduced, because files revised by many developers often contain different design thoughts and coding styles.
	AGE	Average time interval between the last and current change [77]	The lower the AGE, *i.e.*, the more recent the last change, the more likely a defect is introduced.
	NUC	Number of unique changes to the modified files [34], [71]	The larger the NUC, the more likely a defect is introduced, because a developer will have to recall and track many previous changes.
Experience	EXP	Developer experience [7]	More experienced developers are less likely to introduce a defect.
	REXP	Recent developer experience [7]	A developer that has often modified the files in recent months is less likely to introduce a defect, because he will be more familiar with the recent developments in the system.
	SEXP	Developer experience on a subsystem [7]	Developers that are familiar with the subsystems modified by a change are less likely to introduce a defect.

III. APPROACH

In this section, we introduce our new defect predictor NF that leverages the feature representation learning and defect classifier training to learn better feature representations.

A. OVERALL FRAMEWORK

Our model mainly consists of four steps: (i) extracting the hand-crafted features from the labeled changes in the training set; (ii) normalizing all the extracted features and generating a subset using random under-sampling; (iii) training NF using the subset; (iv) using NF to predict the unlabeled changes in the test set to be buggy or clean.

To build our model, we first introduce a decision forest that has a probabilistic routing at each split node as the classifier. Specifically, each split node in this forest has a probability to go left or right. The probability of arriving at a leaf node for an input is the product of all probabilities along the corresponding routing path from the root node to this leaf node. Next, we design a deep neural network for learning feature representations. For each change, this deep network takes the normalized hand-crafted features as inputs and outputs a high-dimensional deep feature vector. Then, a novel model, NF, is built by integrating the representation learning neural network and the decision forest. With above definitions, the output neurons of the neural network are connected to the split nodes of the decision forest, which means the sigmoid transformation results for the output feature vector of the deep network serve as the split probabilities for the split nodes in the decision forest. At last, for an input change, the predicted probability distribution at each leaf node equals to the probability distribution at this leaf node multiplying the arriving probability to this node. The final predicted probability distribution is the sum of predicted probability distributions at all leaf nodes. Therefore, with the input of some hand-crafted features, NF can classifying this change to be buggy or clean by thresholding the final predicted probability distribution. The probability distributions at leaf nodes can be learned by iterative updates. All parameters of the deep neural network and decision forest are optimized using back-propagation by minimizing a global loss function. More details will be provided in Section III-C.

B. FEATURE EXTRACTION

In this study, the 14 hand-crafted features shown in Table 1 are the same with [9], [16]–[18]. These features can be categorized into five dimensions including *diffusion* (*i.e.*, NS, ND, NF, and Entropy), *size* (*i.e.*, LA, LD, and LT), *purpose* (*i.e.*, FIX), *history* (*i.e.*, NDEV, AGE, and NUC), and *experience* (*i.e.*, EXP, REXP, and SEXP). The *diffusion* dimension is based on the distribution of a change. A highly distributed change is more complex to be understood and thus more likely to be defect-inducing [9]. The *size* dimension is based on that the software size is related to defect proneness [72], [73]. Intuitively, if a change is related to more code, the possibility of this change being buggy is higher. The *purpose* dimension can be explained that a change that aims to fix a defect is more likely to introduce a new defect [75]. The *history* dimension is designed to reflect the fact that a change is more likely to be defect-inducing if the files touched by this change have been modified by more developers or by more changes [76]. The *experience* dimension is used to reflect that developers with more experience are less likely to introduce new defects [7]. Otherwise, if the developer experience of a change is less, this change has a higher possibility of being a new defect. Please refer to [9] for the detailed information of these hand-crafted features.

We first process data using logarithmic transformation to make the features in the same order of magnitude. Following

Algorithm 1: Training of NF

Input:
$\mathcal{S}$: Training set of changes, $S \subset \mathcal{X} \times \mathcal{Y}$
Max-Iters: Maximum number of training iterations
SGD-Epochs: Number of epochs to update Θ
RHO-Epochs: Number of epochs to update ρ

Output:
NF: The final NF model trained by $\mathcal{S}$

```
function TrainNF
    Normalize features X in S as in Section III-B;
    Use random under-sampling to get class-balanced S;
    for i = 1 to Max-Iters do
        for j = 1 to RHO-Epochs do
            Update ρ by iterating (13);
        for j = 1 to SGD-Epochs do
            Break data S into random mini-batches;
            forall mini-batches B from S do
                Update Θ using (9) - (12);
    return The NF model with parameters Θ and ρ
```

[9], [16]–[18], we perform a standard log transformation to each feature, except "FIX", which is a binary feature. Moreover, we need not perform feature selection, *e.g.*, removing highly correlated features, which is important in [9], [68], [78]–[80]. To solve the class imbalance problem, we adopt random under-sampling (RUS) on the raw data to improve classification performance. In defect prediction, since the number of clean changes is much more than that of buggy changes, RUS can remove some clean changes randomly, which leads to approximately equal numbers of clean changes and buggy changes.

C. THE NF MODEL

1) The Construction of the NF Model

Fig. 1 shows the architecture diagram of the proposed NF. Suppose our target is to classify a feature vector x ($x \in \mathcal{X}$) into the category y ($y \in \mathcal{Y}$), in which $\mathcal{X}$ is the input feature space and $\mathcal{Y}$ is the output category space. Here, each feature vector x denotes the traditional 14-dimensional change features in Table 1, and the predicted category y is a binary variable with one meaning a buggy change and zero meaning a clean change.

We first introduce a new probabilistic decision forest. In a standard decision tree that has the set $\mathcal{D}$ of internal decision nodes (or split nodes) and the set $\mathcal{T}$ of terminal leaf nodes, an input x is sent to left or right subtree at each node $d \in \mathcal{D}$ until it reaches a leaf node. Each terminal leaf node $t \in \mathcal{T}$ holds a probability distribution ρ_t for all target categories. In this paper, we use a variant of decision forest, probabilistic forest. In a probabilistic tree, the input x has a probabilistic routing to left and right subtrees at each decision node. The probabilistic routing direction obeys Bernoulli distribution with $\phi_d(x|\Theta)$ (decision function) representing the probability of routing to the left subtree of node d, in which Θ denotes forest parameters. Hence the probability of routing to right is $(1-\phi_d(x|\Theta))$. We can formulate the predicted probability for input sample x from tree F as

$$P_F(y|x,\Theta,\rho) = \sum_{t\in\mathcal{T}} \rho_{ty}\varpi_t(x|\Theta), \tag{1}$$

$$\varpi_t(x|\Theta) = \prod_{d\in\mathcal{D}} \phi_d(x|\Theta)^{1[t\in\widehat{d}_l]}(1-\phi_d(x|\Theta))^{1[t\in\widehat{d_r}]}. \tag{2}$$

Here, $\varpi_t(x|\Theta)$ denotes the probability of input's arrival to terminal leaf node t. ρ_{ty} is the probability to take category y at node $t \in \mathcal{T}$. Clearly, we have $\sum_y \rho_{ty} = 1$ and $\sum_{t\in\mathcal{T}} \varpi_t(x|\Theta) = 1$. $1[\cdot]$ is an indicator function who has the value 1 if the condition is true, otherwise has the value 0. $\widehat{d}_l$ and $\widehat{d}_r$ represent the node sets of left subtree and right subtree, respectively. For (2), it is not necessary for a leaf node t to traverse all internal nodes in the implementation. Only the routing path from the root node to the leaf node t along a tree should be taken into consideration. Due to the property of decision tree, we have $|\mathcal{D}| = |\mathcal{T}| - 1$, and the depth of the decision tree is $log(|\mathcal{T}|)$ ($|\cdot|$ means the number of elements in a set). Hence the time complexity to compute (2) for all leaf nodes is $|\mathcal{T}| \cdot log(|\mathcal{T}|)$.

According to the above analyses, the core of the classifier model is to design a proper form of decision function $\phi_d(x|\Theta)$. In the traditional study of decision forests, various of ϕ_d functions have been proposed such as the well-known CART algorithm [81], ID3 algorithm [82], and C4.5 algorithm [83]. Instead of hand-crafted decision functions, in this paper, we aim at learning ϕ_d automatically using the neural network. To achieve this goal, we connect the output neurons of a carefully designed neural network to the internal decision nodes of the decision forest. The routing decision can be thus directly learned by the network. In our design, ϕ_d can be written as

$$\phi_d(x|\Theta) = \sigma(f_d(x;\Theta)), \tag{3}$$

$$\sigma(z) = \frac{1}{1+e^{-z}}, \tag{4}$$

in which $f_d(x;\Theta)$ is an output value of the neural network, and $\sigma(z)$ is a sigmoid function whose value range is $(0,1)$. Here, ϕ_d is designed to convert the learned features of the neural network into the probabilities of routing directions. Hence Θ is the network's parameters that can be learned. Note that the probability distribution ρ_t of each leaf node can also be trained. By this way, we integrate the neural network and decision forest into a holistic model.

For the configuration of the neural network shown in the top part of Fig. 1, we use a four-layer network: an input layer, two hidden layers, and an output layer. Except for the input layer, all of the other three layers are fully connected layers, each of which follows a nonlinearity function (ReLU [84] for two hidden layers and sigmoid for the output layer). The number of neurons in the input layer (N_1) is fixed, *i.e.*, 14 that equals to the number of dimensions of traditional change

TABLE 2. Description of datasets.

Project	Short Name	Period	#Changes	Buggy	Average LOC		# of modified files per change	# of changes per day	# dev. per file	
					File	Change			Max	Avg
Bugzilla	BUG	08/1998-12/2006	4,620	36%	389.8	37.5	2.3	1.5	37	8.4
Columba	COL	11/2002-07/2006	4,455	31%	125.0	149.4	6.2	3.3	10	1.6
Eclipse JDT	JDT	05/2001-12/2007	35,386	14%	260.1	71.4	4.3	14.7	19	4.0
Mozilla	MOZ	01/2000-12/2006	98,275	5%	360.2	106.5	5.3	38.9	155	6.4
Eclipse Platform	PLA	05/2001-12/2007	64,250	14%	231.6	72.2	4.3	26.7	28	2.8
PostgreSQL	POS	07/1996-05/2010	20,431	25%	563.0	101.3	4.5	4.0	20	4.0

features. The numbers of neurons in the two hidden layers (N_2, N_3) are adjustable. The number of neurons in the output layer (N_4) will be introduced in the following. The *universal approximation theorem* [85] states that simple neural networks with a single hidden layer that contains a finite number of neurons, and with arbitrary activation functions under mild assumptions, are universal approximators for the continuous functions on compact subsets of R^n when given appropriate parameters. According to this theorem, our NF model has the potential ability to represent the optimal features for defect prediction, because NF has two hidden layers.

A forest can be viewed as an ensemble of multiple decision trees, which can be formulated as $\mathcal{F} = \{F_1, F_2, \ldots, F_{|\mathcal{F}|}\}$ where $|\mathcal{F}|$ is the number of trees. The final prediction result of our model NF is the average of all decision trees. Therefore, the probability of an input sample x to take on category y is

$$P_{\mathcal{F}}(y|x, \Theta, \rho) = \frac{1}{|\mathcal{F}|} \sum_{m=1}^{|\mathcal{F}|} P_{F_m}(y|x, \Theta, \rho). \tag{5}$$

Since the output layer of the neural network is connected to the internal decision nodes, the number of its neurons can be computed by

$$N_4 = |\mathcal{F}| \cdot |\mathcal{D}| = |\mathcal{F}| \cdot (|\mathcal{T}| - 1). \tag{6}$$

Hence N_4 is determined by the number of decision trees in the forest and the depth of each decision tree.

With the above definitions, we can input an unlabeled software change into the proposed NF model following the above steps to predict it to be buggy or clean. Since our model combines the powerful abilities of the neural network's representation and the decision forest's classification, we call our model *neural forest* (**NF**). In this holistic model, the neural network can generate proper features for consequent decision forest, and the decision forest can guide the learning process of the neural network.

2) The Training of NF

Now, we present the training process of our NF model. The global loss function for a training sample (x, y) is

$$L(\Theta, \rho; x, y) = -log P_{\mathcal{F}}(y|x, \Theta, \rho). \tag{7}$$

Our training goal is to minimize this loss function. To optimize Θ and ρ, we use an alternate update.

We first update Θ with fixed ρ. Suppose we have a training set of $\mathcal{S} \subset \mathcal{X} \times \mathcal{Y}$. Given a random subset $\mathcal{B}$ (*i.e.*, mini-batch) of samples from the training set $\mathcal{S}$, the total loss on batch $\mathcal{B}$ can be written as

$$J(\Theta, \rho; \mathcal{B}) = \frac{1}{|\mathcal{B}|} \sum_{(x,y)\in\mathcal{B}} L(\Theta, \rho; x, y) + \frac{\lambda}{2}||\Theta||_2^2. \tag{8}$$

The second term is a regularization term (also called a weight decay term) whose purpose is to reduce the magnitudes of the weights and prevent overfitting. The weight decay parameter λ is used to balance the importance of these two terms. We can use Stochastic Gradient Descent (SGD) to minimize (8) as:

$$\begin{aligned}\Theta^{(h+1)} &= \Theta^{(h)} - \alpha \frac{\partial J(\Theta^{(h)}, \rho; \mathcal{B})}{\partial \Theta} \\ &= \Theta^{(h)} - \alpha [\frac{1}{|\mathcal{B}|} \sum_{(x,y)\in\mathcal{B}} \frac{\partial L(\Theta^{(h)}, \rho; x, y)}{\partial \Theta} + \lambda \Theta^{(h)}],\end{aligned} \tag{9}$$

in which $\Theta^{(h)}$ is the Θ at time h and α is the learning rate. The key of (9) is to compute the partial derivative of $L(\Theta, \rho; x, y)$ with respect to network weights Θ. We can decompose it using the chain rule as

$$\frac{\partial L(\Theta, \rho; x, y)}{\partial \Theta} = \sum_{d\in\mathcal{D}} \frac{\partial L(\Theta, \rho; x, y)}{\partial f_d(x; \Theta)} \cdot \frac{\partial f_d(x; \Theta)}{\partial \Theta}. \tag{10}$$

$\frac{\partial f_d(x;\Theta)}{\partial \Theta}$ can be computed as commonly done in the context of neural networks. Considering a forest containing a single tree (*i.e.*, $\mathcal{F} = \{F\}$), we have

$$\begin{aligned}\frac{\partial L(\Theta, \rho; x, y)}{\partial f_d(x; \Theta)} &= \sum_{t\in\mathcal{T}} \frac{\partial L(\Theta, \rho; x, y)}{\partial \varpi_t(x|\Theta)} \cdot \frac{\partial \varpi_t(x|\Theta)}{\partial f_d(x; \Theta)} \\ &= -\sum_{t\in\mathcal{T}} \frac{\rho_{ty}}{P_F(y|x, \Theta, \rho)} \cdot \frac{\partial \varpi_t(x|\Theta)}{\partial f_d(x; \Theta)} \\ &= -\sum_{t\in\mathcal{T}} \frac{\rho_{ty} \varpi_t(x|\Theta)}{P_F(y|x, \Theta, \rho)} \cdot \frac{\partial log \varpi_t(x|\Theta)}{\partial f_d(x; \Theta)},\end{aligned} \tag{11}$$

and thus

$$\begin{aligned}\frac{\partial L(\Theta, \rho; x, y)}{\partial f_d(x; \Theta)} = \sum_{t\in\mathcal{T}} \frac{\rho_{ty} \varpi_t(x|\Theta)}{P_F(y|x, \Theta, \rho)} (&1[t \in \widehat{d}_r] \cdot \phi_d(x|\Theta) \\ &- 1[t \in \widehat{d}_l] \cdot (1 - \phi_d(x|\Theta))).\end{aligned} \tag{12}$$

Hence we can optimize Θ using back-propagation iteratively.

With fixed Θ, we follow [86]–[88] to iteratively update ρ as follows

$$\rho_{ty}^{(h+1)} = \frac{1}{Z_t^{(h)}} \sum_{(x,y')\in\mathcal{S}} \frac{1[y = y'] \rho_{ty}^{(h)} \varpi_t(x|\Theta)}{P_F(y|x, \Theta, \rho^{(h)})}, \tag{13}$$

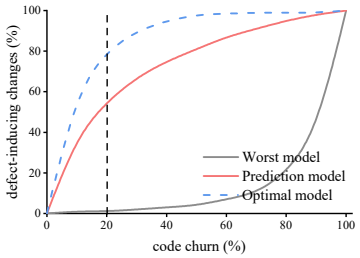


FIGURE 2. Code-churn-based Alberg diagram.

in which $1/Z_t^{(h)}$ is a normalization term ensuring $\sum_{y\in\mathcal{Y}} \rho_{ty}^{(h+1)} = 1$. We set $\rho_{ty}^{(0)} = 1/|\mathcal{Y}|$. Instead of mini-batches that are used to update Θ, we use the whole training set for the optimization of ρ. The training of NF is summarized in Algorithm 1.

In the test phase, each NF model can output a probability value of the change being defect-inducing. Following [16]–[18], a change is viewed as a defect if its defect probability value is larger than 0.5.

IV. EXPERIMENTAL SETUP

A. BENCHMARKS AND BASELINES

Benchmarks. We perform our experiments on six large-scale projects (*i.e.*, Bugzilla, Columa, Eclipse JDT, Eclipse Platform, Mozilla, and PostgreSQL), which are widely used in the field of effort-aware JIT-SDP [9], [16]–[19]. Table 2 describes the statistics of these datasets, including their names, the period ranges, the total number of changes in each project, the percentage of defect-inducing changes for each project, the average numbers of LOCs at the file level and the change level, the average number of modified files per change, and the average number of the changes per day. Table 2 also shows the maximum and average numbers of the developers that modified a single file. From Table 2, we can see that each of these projects has lasted for a long period, *i.e.*, ranging from 4 years to 14 years. The numbers of changes for these six projects are in the range of 4,455 - 98,275. Moreover, we can see that the percentages of buggy changes are in the range of 5% - 36%, and it is easy to find that all the datasets are imbalanced. The diversity of these datasets makes the experimental results more representative.

Previous state-of-the-art models. We compare our NF with four most recent state-of-the-art effort-aware JIT-SDP models which are EALR model proposed by Kamei *et al.* [9], LT model proposed by Yang *et al.* [16], OneWay model proposed by Fu *et al.* [18], and CBS model proposed by Huang *et al.* [17]. Among these competitors, EALR and CBS are supervised predictors, LT is an unsupervised predictor, and OneWay is a supervised model built on the implication of multiple simple unsupervised models proposed by Yang *et al.* [16]. All the baselines use the released code provided by the authors with default settings.

Baselines. We design five baselines to compare with our NF to demonstrate the benefit of the holistic system based on neurons and forest. The brief description of these baselines are: (i) The first baseline directly uses hand-crafted features and then apply **random forest (RF)** for classification. (ii) The second one also uses hand-crafted features and then apply **SVM** for classification. (iii) The third and fourth models first apply two widely used feature selection techniques, *i.e.*, Maximal Information Coefficient (MIC) [89] and CHI-Square Statistics (CHI) [58], to identify a subset of representative features, respectively. Then a random forest (RF) is used as the classifier. These two baselines are denoted as **M+RF (MIC+RF)** and **C+RF (CHI+RF)**, respectively. (iv) The fifth baseline employs a deep neural network to learn new feature representations from hand-crafted features. Then, a simple **Softmax (SM)** function is used for classification, which is a traditional function in the deep neural network. Compared with these baselines, we can further show the superiority of the proposed NF. For examples, by comparing with two widely used feature selection techniques, we can show the powerful ability of NF in automatic feature exploration. For all baselines, we have used their best parameter settings.

TABLE 3. Value ranges of Cliff's δ and the corresponding effectiveness levels [96].

Cliff's δ	Effectiveness Level
$-1 \leq \delta < 0.147$	Negligible
$0.147 \leq \delta < 0.33$	Small
$0.33 \leq \delta < 0.474$	Medium
$0.474 \leq \delta \leq 1$	Large

Evaluation metrics. To evaluate the performance of various JIT-SDP models, we use five evaluation metrics: recall, precision, F1-score, and P_{opt}, all of which are based on effort-aware evaluations. Here, we follow Kamei *et al.* [9] to use the number of the lines of code (LOC) related to a change to measure the effort to inspect this change. Recall, precision, F1-score, and AUC are four well-known metrics [90]. Recall refers to the ratio of the number of instances, which are correctly classified as buggy changes when inspecting 20% of the total number of LOC modified by all changes, to the total number of actual buggy instances. Precision refers to the ratio of the number of instances correctly classified to be buggy to the total number of instances classified to be buggy when inspecting 20% of the total number of LOC modified by all changes. Considering both recall and precision, F1-score is the weighted harmonic average of them. F1-score can be computed by

$$F1\text{-}score = \frac{2 \times Recall \times Precision}{Recall + Precision}. \quad (14)$$

AUC [91], [92] is a trade-off between true positive rate (TPR) and false positive rate (FPR). AUC is robust to imbalance class distribution and misclassification costs, so it is widely used as an evaluation measure for defect prediction [93]–[95].

TABLE 4. Comparison among our NF model, previous state-of-the-art models, and the designed five baselines in term of aforementioned five metrics for within-project effort-aware JIT-SDP. The p-value and Cliff's δ are calculated using the results on all datasets when comparing NF with other methods. The best result for each dataset under each metric is highlighted in **red**.

Data	Metrics	EALR	LT	OW	CBS	RF	SVM	M+RF	C+RF	SM	NF
BUG	Recall	0.413	0.430	0.435	0.567	0.449	0.427	0.422	0.419	0.547	**0.612**
	Precision	0.387	0.279	0.391	0.501	0.605	**0.650**	0.618	0.639	0.547	0.606
	F1-score	0.396	0.338	0.408	0.532	0.507	0.503	0.496	0.505	0.541	**0.602**
	AUC	0.322	-	-	-	0.522	0.512	0.506	0.500	0.634	**0.641**
	P_{opt}	0.723	0.730	0.752	0.731	0.918	0.935	0.916	0.880	0.944	**0.956**
COL	Recall	0.409	0.621	**0.680**	0.525	0.432	0.303	0.379	0.418	0.543	0.568
	Precision	0.333	0.240	0.273	0.460	0.544	**0.621**	0.595	0.558	0.450	0.535
	F1-score	0.344	0.344	0.387	0.490	0.459	0.364	0.425	0.464	0.491	**0.549**
	AUC	0.374	-	-	-	0.500	0.511	0.500	0.500	0.611	**0.628**
	P_{opt}	0.599	0.838	0.852	0.618	0.846	0.849	0.900	0.900	0.916	**0.930**
JDT	Recall	0.222	0.576	0.576	0.543	**0.710**	0.674	0.607	0.697	0.592	0.561
	Precision	0.166	0.115	0.115	0.239	0.221	0.222	0.239	0.219	0.254	**0.335**
	F1-score	0.169	0.191	0.191	0.331	0.337	0.334	0.340	0.333	0.352	**0.414**
	AUC	0.370	-	-	-	0.592	0.597	0.576	0.593	0.603	**0.625**
	P_{opt}	0.494	0.798	0.798	0.650	0.887	0.899	0.912	0.885	0.894	**0.922**
MOZ	Recall	0.144	0.365	0.365	0.445	0.618	0.621	0.604	0.560	**0.654**	0.581
	Precision	0.052	0.033	0.033	0.109	0.127	0.128	0.128	0.138	0.118	**0.202**
	F1-score	0.076	0.061	0.061	0.175	0.208	0.211	0.210	0.220	0.198	**0.289**
	AUC	0.358	-	-	-	0.643	0.630	0.632	0.633	0.652	**0.665**
	P_{opt}	0.463	0.658	0.658	0.624	0.948	0.949	0.941	0.948	0.904	**0.951**
PLA	Recall	0.300	0.491	0.491	0.611	0.778	0.777	0.485	**0.789**	0.563	0.610
	Precision	0.194	0.114	0.114	0.247	0.213	0.215	0.277	0.218	0.230	**0.342**
	F1-score	0.223	0.185	0.185	0.352	0.334	0.336	0.350	0.341	0.318	**0.426**
	AUC	0.348	-	-	-	0.615	0.618	0.500	0.614	0.602	**0.637**
	P_{opt}	0.551	0.759	0.759	0.709	0.912	0.909	0.915	0.914	0.923	**0.944**
POS	Recall	0.289	0.536	0.536	0.501	0.512	0.517	0.574	0.516	0.620	**0.651**
	Precision	0.244	0.185	0.185	0.447	**0.549**	0.541	0.495	0.545	0.413	0.508
	F1-score	0.258	0.274	0.274	0.472	0.526	0.525	0.525	0.522	0.493	**0.566**
	AUC	0.342	-	-	-	0.623	0.603	0.636	0.571	0.657	**0.681**
	P_{opt}	0.532	0.792	0.792	0.627	0.911	0.909	0.906	0.918	0.923	**0.944**
AVE	Recall	0.296	0.503	0.514	0.532	0.583	0.553	0.512	0.567	0.587	**0.597**
	Precision	0.229	0.161	0.185	0.334	0.377	0.396	0.392	0.386	0.335	**0.421**
	F1-score	0.244	0.232	0.251	0.392	0.395	0.379	0.391	0.398	0.399	**0.474**
	AUC	0.352	-	-	-	0.582	0.579	0.558	0.569	0.627	**0.646**
	P_{opt}	0.559	0.763	0.769	0.660	0.904	0.908	0.915	0.908	0.917	**0.941**
p-value	Recall	0.0156	0.0465	0.0660	0.0205	0.4686	0.4686	0.0460	0.1970	0.3496	-
	Precision	0.0205	0.0043	0.0129	0.1548	0.4686	0.5909	0.4091	0.5314	0.2424	-
	F1-score	0.0043	0.0043	0.0043	0.1548	0.1548	0.0898	0.1201	0.1548	0.1548	-
	AUC	0.0010	-	-	-	0.0129	0.0043	0.0151	0.0064	0.1201	-
	P_{opt}	0.0024	0.0024	0.0024	0.0024	0.0151	0.0323	0.0064	0.0151	0.0217	-
Cliff's δ	Recall	1(L)	0.611(L)	0.556(L)	0.722(L)	0.056(N)	0.056(N)	0.556(L)	0.333(M)	0.167(S)	-
	Precision	0.722(L)	0.889(L)	0.778(L)	0.389(M)	0.006(N)	-0.556(N)	0.111(N)	0(N)	0.228(S)	-
	F1-score	0.889(L)	0.889(L)	0.889(L)	0.389(M)	0.389(M)	0.500(L)	0.474(L)	0.389(M)	0.389(M)	-
	AUC	1(L)	-	-	-	0.778(L)	0.889(L)	0.778(L)	0.889(L)	0.474(L)	-
	P_{opt}	1(L)	1(L)	1(L)	1(L)	0.778(L)	0.667(L)	0.889(L)	0.778(L)	0.722(L)	-

Moreover, P_{opt} is based on the conception of "code-churn-based" Alberg diagram. An example is shown in Fig. 2. Similarly to the definition in previous studies [9], [14], [19], [20], code churn refers to the number of LOC added and deleted by the changes. In this diagram, the x-axis refers to the cumulative percentage of effort needed to inspect the changes, and the y-axis refers to the cumulative percentage of detected defect-inducing changes. This diagram shows the curves of the prediction model, the "optimal" model, and the "worst" model. For the "optimal" model, the actual buggy changes are sorted in ascending order according to their code churn while in decreasing order for the "worst" model. In the prediction model, we first select the changes whose probability value predicted by NF greater than 0.5 and then sort these changes in ascending order according to their *probability density*. The probability density $PD(c)$ of a change c can be formulated as

$$PD(c) = \frac{P(c)}{\mathit{Effort}(c)}, \tag{15}$$

in which $P(c)$ is the predicted probability value of change c being buggy, and $\mathit{Effort}(c)$ is the effort needed to inspect change c. Following [9], [14], [19], [20], the $\mathit{Effort}(c)$ is computed by $\mathit{Effort}(c) = LA(c) + LD(c)$ ($LA(c)$ refers to the number of LOC added by the change c, and $LD(c)$ refers to the number of LOC deleted by the change c).

Based on the definitions by Kamei *et al.* [9], P_{opt} can be computed as

$$P_{opt}(pred) = 1 - \frac{S(optimal) - S(pred)}{S(optimal) - S(worst)}. \tag{16}$$

Here, $S(optimal)$, $S(pred)$ and $S(worst)$ are the areas under the curves of "optimal", prediction and "worst" models,

TABLE 5. Comparison between our NF model and previous state-of-the-art models for cross-project effort-aware JIT-SDP. The p-value and Cliff's δ are calculated using the results on all datasets when comparing NF with other methods. The best result for each dataset under each metric is highlighted in **red**.

Train	Test	EALR		LT		OW		CBS		NF	
		F1	P_{opt}	F1	P_{opt}	F1	P_{opt}	F1	P_{opt}	F1	P_{opt}
BUG	COL	0.352	0.678	0.351	0.858	0.393	0.868	0.420	0.577	**0.478**	**0.904**
	JDT	0.199	0.582	0.190	0.815	0.195	0.769	0.303	0.601	**0.377**	**0.928**
	MOZ	0.066	0.561	0.061	0.660	0.058	0.622	0.145	0.655	**0.231**	**0.993**
	PLA	0.220	0.681	0.186	0.770	0.184	0.740	0.334	0.654	**0.376**	**0.998**
	POS	0.282	0.603	0.270	0.810	0.294	0.762	0.381	0.581	**0.540**	**0.995**
COL	BUG	0.360	0.651	0.364	0.726	0.411	0.758	0.468	0.684	**0.569**	**0.995**
	JDT	0.110	0.406	0.190	0.815	0.195	0.770	0.257	0.584	**0.364**	**0.936**
	MOZ	0.071	0.481	0.061	0.660	0.058	0.622	0.107	0.640	**0.262**	**0.998**
	PLA	0.196	0.509	0.186	0.770	0.184	0.740	0.263	0.537	**0.356**	**0.998**
	POS	0.247	0.507	0.270	0.810	0.294	0.762	0.389	0.611	**0.543**	**0.999**
JDT	BUG	0.373	0.682	0.364	0.726	0.364	0.726	0.506	0.781	**0.611**	**0.993**
	COL	0.327	0.558	0.351	0.858	0.351	0.858	0.461	0.594	**0.503**	**0.899**
	MOZ	0.062	0.500	0.061	0.660	0.061	0.660	0.085	0.772	**0.189**	**0.996**
	PLA	0.184	0.552	0.186	0.770	0.186	0.770	0.325	0.733	**0.405**	**0.999**
	POS	0.270	0.520	0.270	0.810	0.270	0.810	0.391	0.749	**0.556**	**0.996**
MOZ	BUG	0.365	0.649	0.364	0.726	0.364	0.726	0.456	0.542	**0.604**	**0.974**
	COL	0.320	0.502	0.351	0.858	0.351	0.858	0.448	0.482	**0.498**	**0.908**
	JDT	0.104	0.397	0.190	0.815	0.190	0.815	0.334	0.485	**0.385**	**0.922**
	PLA	0.184	0.447	0.186	0.770	0.186	0.770	0.334	0.506	**0.399**	**0.998**
	POS	0.187	0.387	0.270	0.810	0.270	0.810	0.426	0.528	**0.549**	**0.995**
PLA	BUG	0.393	0.693	0.364	0.726	0.364	0.726	0.496	0.678	**0.615**	**0.983**
	COL	0.332	0.547	0.351	0.858	0.351	0.858	0.427	0.547	**0.517**	**0.918**
	JDT	0.152	0.445	0.190	0.815	0.190	0.815	0.325	0.625	**0.411**	**0.915**
	MOZ	0.077	0.497	0.061	0.660	0.061	0.660	0.103	0.713	**0.245**	**0.995**
	POS	0.258	0.503	0.270	0.810	0.270	0.810	0.371	0.673	**0.569**	**0.996**
POS	BUG	0.362	0.639	0.364	0.726	0.364	0.726	0.408	0.491	**0.563**	**0.981**
	COL	0.335	0.606	0.351	0.858	0.351	0.858	0.504	0.638	**0.508**	**0.913**
	JDT	0.145	0.458	0.190	0.815	0.190	0.815	0.313	0.548	**0.377**	**0.885**
	MOZ	0.061	0.495	0.061	0.660	0.061	0.660	0.157	0.579	**0.237**	**0.995**
	PLA	0.166	0.536	0.186	0.770	0.186	0.770	0.320	0.564	**0.389**	**0.999**
AVE		0.225	0.542	0.226	0.773	0.242	0.764	0.342	0.612	**0.441**	**0.967**
p-value		3.2e-08	1.4e-11	4.1e-08	1.3e-11	1.4e-07	1.4e-11	0.0025	1.4e-11	-	-
Cliff's δ		0.813(L)	1(L)	0.806(L)	1(L)	0.772(L)	1(L)	0.421(M)	1(L)	-	-

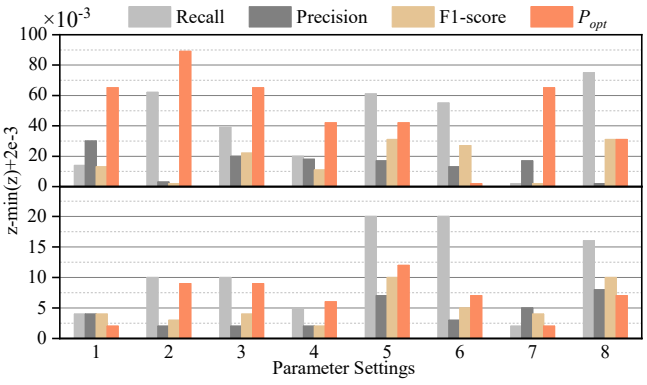
respectively. A higher value of $P_{opt}(pred)$ suggests that the predictor $pred$ is closer to the "optimal" model.

Experimental design. We mainly consider two scenarios, *i.e.*, within-project and cross-project defect prediction. (i) For within-project defect prediction, we perform 10-fold cross-validation to evaluate our model. In particular, we divide each dataset into ten folds with approximately equal sizes. We ensure that the proportion of buggy changes and clean changes in each fold is the same as the original dataset to keep the property of the original benchmark. Then, each fold is used to test the model trained on the other nine folds. Hence we will conduct ten trials for each model on each dataset. These ten evaluation results will be averaged to obtain the final score. (ii) For cross-project defect prediction, we train our model on one project (*i.e.*, source project) and use the trained model to predict the changes on other projects (*i.e.*, target projects).

To statistically test the differences between our NF with the four recent state-of-the-art models and five baselines, we perform Wilcoxon signed-rank test and compute the p-value in the same way as Yang *et al.* [18]. To control the false discovery rate, we further use the Benjamini-Hochberg (BH) adjusted p-value to check if two distributions are statistically significant at the significance level of 0.05 [97]. If the statistical test shows a significant difference, we then adopt the Cliff's δ [96] to measure whether the magnitude of the difference is practically important from the viewpoint of practical application to further verify the effectiveness of our method [98], [99]. Cliff's δ is widely used to judge whether the difference between two variables is significant. As Romano *et al.* [98] suggested, the ranges of Cliff's δ values and the corresponding effectiveness levels are: negligible ($-1 \leq \delta < 0.147$), small ($0.147 \leq \delta < 0.33$), medium ($0.33 \leq \delta < 0.474$), and large ($0.474 \leq \delta \leq 1$), as shown in Table 3. According to the definition of Cliff's δ, NF is significantly better than the baselines if the effectiveness level mapped from the Cliff's δ is "large".

B. NF SETTINGS

According to the properties of decision trees, the numbers of internal nodes and leaf nodes are decided by tree depth as discussed in Section III-C2. The number of trees in the forest ($|\mathcal{F}|$), tree depth, and the numbers of neurons in hidden layers (N_2, N_3) are adjustable. The ablation study for them are presented in Section V-C. In Algorithm 1, we set *Max-Iters* to 50, which means we alternately train Θ and ρ for 50 times. The *SGD-Epochs* and *RHO-Epochs* are set to 10 and 20, respectively. The batch size ($|\mathcal{B}|$) for the update of Θ is set to 300. α is set to 0.001.



(a)

Parameter	1	2	3	4	5	6	7	8
$\|\mathcal{F}\|$	2	2	10	10	5	5	5	5
Depth	5	7	5	5	5	5	5	3
N_2	256	1024	1024	512	1024	512	256	1024
N_3	1024	2048	2048	2048	2048	2048	1024	2048

(b)

FIGURE 3. Ablation studies of using various settings for the adjustable parameters discussed in Section IV-B. (a) **Top:** The evaluation results on the Bugzilla benchmark; **Bottom:** The evaluation results on the JDT benchmark. (b) The corresponding parameter settings for ablation models in the left. Note that the y-axis represents $z - min(z) + 2e$-3 where z is the evaluation results for a metric on a dataset. Best viewed in color.

C. RESEARCH QUESTIONS

To evaluate the effectiveness of NF, we investigate the following three research questions:

- **RQ1:** How well does NF predict defect-inducing changes when compared with the designed baselines and previous state-of-the-art models for within-project effort-aware JIT-SDP?
- **RQ2:** How well does NF predict defect-inducing changes when compared with previous state-of-the-art models for cross-project effort-aware JIT-SDP?
- **RQ3:** What is the effect of varying the parameter settings and reducing the amount of training data for NF?

The first two research questions are designed to compare our NF with previous state-of-the-art prediction models and the five designed baselines in term of five metrics mentioned above for within-project and cross-project effort-aware just-in-time defect prediction, respectively. To answer RQ3, we first adjust parameters of NF and explore the influence of various parameter settings for defect prediction. This part explains the selected parameters for our model. Besides, for some machine learning models, the limited training data will restrict the final performance. Thus, we also explore the effectiveness of NF when reducing the amount of training data.

V. EXPERIMENTAL RESULTS

In this section, we present the experimental results to answer the aforementioned research questions.

A. WITHIN-PROJECT EFFORT-AWARE JIT-SDP (RQ1)

The recall, precision, F1-score, AUC and p_{opt} of NF model, four previous models and five designed baselines are summarized in Table 4. Since LT, OW, and CBS can only output the binary classification for input changes, we cannot compute the AUC metric for them. The corresponding positions in Table 4 are filled with the symbol "-". We can see that the proposed NF outperforms all competitors on all datasets.

Comparison with state-of-the-art models. First, NF can achieve high recall values ranging from ∼56% to ∼65%. On average, NF can achieve the recall of 59.7%, which is 30.1%, 9.4%, 8.3% and 6.5% higher than EALR, LT, OW, and CBS, respectively. Second, the precision values of NF are 60.6%, 53.5%, 33.5%, 20.2%, 34.2% and 50.8% on all six databases, and the average precision is 42.1%, which means NF can predict buggy changes more precisely. Third, NF can achieve an average F1-score of 47.4%, which is 23.0%, 24.2%, 22.3% and 8.2% higher than EALR, LT, OW, and CBS, respectively. We can also see that the F1-score of our designed baselines are almost as good as CBS that is the most competitive previous model.

Moreover, in terms of a stable metric of AUC, NF achieves significantly higher results than other baselines. The metric of P_{opt} reflects the difference between a designed predictor and the ideal "optimal" model. Higher P_{opt} values demonstrate that NF is very closer to the "optimal" model. The average P_{opt} value of NF is 94.1%, while the average P_{opt} values of EALR, LT, OW and CBS are 55.9%, 76.3%, 76.9% and 66.0%, respectively.

Comparison with designed baselines. Compared with RF and SVM models, the improvement of NF demonstrates its effectiveness in automatic feature exploration, so that the learned feature representations of NF are more discriminative than the original features. Compared with M+RF and C+RF, the improvement of NF proves it can do better in feature representation learning than previous feature selection techniques. We believe it is because of the powerful representation capability of deep neural networks. Through comparison with SM, we prove the importance of the holistic learning of the neural network and forest. Since the learned decision forest is more powerful than a simple nonlinear activation, the proposed NF model can perform better than a simple neural network with Softmax activation.

The bottom lines of Table 4 present the adjusted p-value and Cliff's δ when comparing NF with nine competitors in terms of all metrics for within-project effort-aware JIT-SDP.

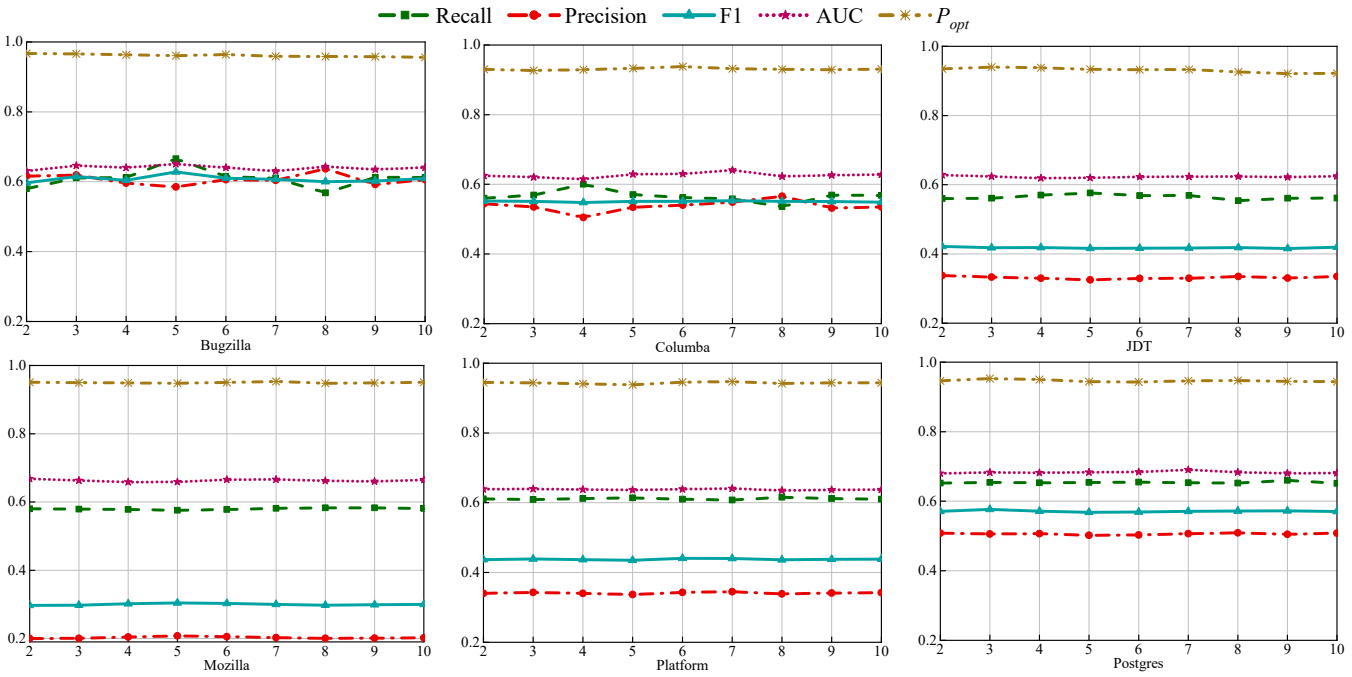


FIGURE 4. The evaluation results of NF's 2-fold to 10-fold experiments on six datasets.

We can observe that most of the p-values are smaller than 0.05 and the effectiveness of NF is thus significant. Moreover, most Cliff's δ values reach the "large" effectiveness level with $\delta \geq 0.474$. This demonstrates that the improvement of NF is nontrivial in terms of all metrics when compared with previous methods or simple combinations of machine learning models.

B. CROSS-PROJECT EFFORT-AWARE JIT-SDP (RQ2)

In this part, we compare NF with previous models for cross-project effort-aware just-in-time defect prediction, which means the training set and test set are from different datasets. In Table 5, we display the F1-score and P_{opt} of NF and four state-of-the-art models. Here, we only report results for F1-score and P_{opt}, because F1-score can reflect the metrics of recall, precision, and AUC. Due to the limitation of space, we do not compare with the designed baselines here. In Table 5, we highlight the top results in red. We can see that all of these highlights lie in the column of NF. It demonstrates that our NF significantly outperforms other methods in terms of all metrics. In the following paragraphs, we summarize the comparison between our model and other competitors for each metric in detail.

Comparing NF with other baselines in terms of F1-score, NF can achieve an average F1-score of 44.1%, which is 21.6%, 21.5%, 19.9% and 9.9% higher than EALR [9], LT [16], OneWay [18], and CBS [17], respectively. Hence we can conclude that NF significantly outperforms all the baselines in terms of the F1-score metric.

Moreover, from the evaluation results of NF and other methods in terms of P_{opt}, it is easy to find that our NF significantly outperforms other methods. On average, NF can achieve 96.7% in term of P_{opt} when inspecting only 20% of the LOC modified by all changes. The average P_{opt} values of the EALR [9], LT [16], OneWay [18], and CBS [17] are 54.2%, 77.3%, 76.2%, and 61.2%, respectively. For the previous state-of-the-art method of LT, the average P_{opt} value of NF is 19.4% higher than its. The significant improvement demonstrates the effectiveness of our method.

The bottom lines of Table 5 present the adjusted p-value and Cliff's δ when comparing NF with nine competitors in terms of F1-score and P_{opt} for cross-project effort-aware JIT-SDP. We can see that all p-values are much smaller than 0.05 and almost all Cliff's δ values fall into the "large" effectiveness level. Therefore, NF is significantly better than all competitors for cross-project effort-aware JIT-SDP. Overall, the above observations suggest that our model is general to cross-project defect prediction and performs better than state-of-the-art models under the cross-project setting.

C. VARIOUS PARAMETER SETTINGS AND VARIOUS AMOUNT OF TRAINING DATA (RQ3)

To evaluate the effect of various settings for the adjustable parameters discussed in Section IV-B, we perform eight groups of ablation studies for the cross-validation of each adjustable parameter in NF. Since the numbers of changes in Bugzilla and JDT datasets are representative among all six datasets, we perform experiments on these two datasets. The evaluation results and the corresponding parameter settings are displayed in Fig. 3. To best visualize the difference, for each metric on each dataset, we subtract the minimum of all results and add $2e$-3 for each result. We can see the fifth group

TABLE 6. Impact factors of 14 input features on six datasets. The top three results are marked in **red**, while the worst three are marked in **blue**.

Features	Diffusion				Size			Purpose	History			Experience		
	NS	ND	NF	Entropy	LA	LD	LT	FIX	NDEV	AGE	NUC	EXP	REXP	SEXP
BUG	0.059	**0.094**	0.050	0.082	0.045	**0.027**	**0.033**	0.058	0.048	**0.318**	**0.083**	0.041	0.040	**0.023**
COL	**0.046**	**0.111**	0.068	0.068	0.052	0.075	**0.034**	0.048	**0.076**	0.071	**0.216**	0.058	0.047	**0.030**
JDT	0.070	**0.161**	0.090	0.064	**0.118**	**0.037**	**0.008**	0.089	**0.026**	0.100	**0.106**	0.040	0.053	0.038
MOZ	0.065	0.096	**0.160**	**0.137**	0.044	0.028	0.085	**0.023**	**0.129**	0.033	0.121	**0.023**	0.038	**0.018**
PLA	0.037	**0.160**	0.096	0.105	0.100	0.041	0.050	**0.123**	**0.031**	0.052	**0.125**	**0.033**	**0.013**	**0.033**
POS	**0.129**	**0.176**	0.101	0.045	0.108	0.031	**0.001**	**0.023**	0.055	**0.160**	0.088	0.040	0.030	**0.014**
AVE	0.068	**0.133**	0.094	0.084	0.078	0.040	**0.035**	0.061	0.061	**0.122**	**0.123**	0.039	**0.037**	**0.026**

of parameters achieves the best performance on the JDT dataset and a good trade-off across all metrics on the Bugzilla dataset. Hence we adopt the fifth group of parameters as the default setting for NF. All experiments in this paper follow this setting.

In the above experiments, we perform 10-fold cross-validations, which means in each trial, 90% of the data are used for training, and another 10% of the data are used for testing. However, some defect predictors are sensitive to the amount of training data, and the limited training data will lead to restrictive performance. To evaluate the proposed NF under the condition that only limited training data are available, we perform 2-fold to 10-fold cross-validations, in which 2-fold means only 50% of the data are used for training. Fig. 4 shows the results on all six datasets. The proposed NF model performs stably in terms of all evaluation metrics. For example, on the JDT benchmark, the largest fluctuations for recall, precision, F1-score and P_{opt} are less than 2.7%, 1.1%, 0.8% and 2.0%, respectively. Even on the Bugzilla dataset that has much less training data than other datasets, the performance of NF is stable enough. Therefore, we can come to the conclusion that NF is stable to the reduced amount of training data.

VI. DISCUSSION

Taking effort-aware JIT-SDP as an example, in this section, we show how to use the NF model to analyze the contribution of the given hand-crafted features to prediction performance.

A. WHAT ARE THE MAJOR FEATURES OF DEFECT-INDUCING CHANGES?

To better understand the influence of each of the given features in defect prediction, we design an experiment to measure the importance of various features. Intuitively, a feature contributing more to the network inference results will have a more significant influence. Because all input features have been limited into the same value range by zero-mean normalization, the contribution of each feature can be measured by the corresponding network weights. Specifically, we compute the impact factor of a feature by summarizing the absolute values of the weights corresponding to this feature, between the first layer (input features) and the second layer of the proposed network. Then, we normalize all impact factor into the range of $[0, 1]$ for easier observation.

Table 6 displays the impact factors of 14 features on all datasets. We can see that the number of modified directories (ND) is the most important feature of the defect-inducing changes on most datasets (except Mozilla). The time when the files were last changed (AGE) and the number of unique last changes of the modified files (NUC) are also important, risk-increasing features. Moreover, the diffusion features (NF and Entropy) and the relative size feature (LA) have considerable influence on the performance of defect prediction. Therefore, the *diffusion* features and *history* features are the most major dimensions which developers should take notice of when they review the code. The reason why *diffusion* features are important is that highly distributed changes are more complex to be understood and thus more likely to be defect-inducing [9]. The reason why *history* features are essential is that a change is more likely to be defect-inducing if the files touched by this change have been modified by more developers or by more changes [76]. On the contrary, the relative size features (LD and LT) and the developer experience features (EXP, REXP, and SEXP) are less critical.

B. WHICH FEATURES HAVE GREAT IMPACT ON A SPECIFIC METRIC?

It is widely accepted that recall, precision and F1-score can measure the false negative (*i.e.*, the defect-inducing changes that are not classified as defect-inducing) and the false positive (*i.e.*, the non-defect-inducing changes that are classified as defect-inducing). To further understand which features have a high impact on each metric (*i.e.*, recall, precision, and F1-score), at each time of testing, we remove a specific feature to measure the changes in terms of each metric compared with original performance. If the removing of a specific feature leads to performance degradation on a metric, this feature has a positive impact on this metric; and vice versa.

As shown in Table 7, when we remove the features FIX, NUC and SEXP, the performance degradation in terms of recall is larger than other features. In terms of precision, the degradation is the largest when the features NUC, AGE, and ND are removed. Therefore, the features FIX, NUC and SEXP contribute most to recall, and the features NUC, AGE, and ND contribute most to precision. The developers can consider specific features under different demands. For example, if we want to improve precision, *i.e.*, selecting defect-inducing changes from the changes classified as non-defect-inducing, we can mainly analyze the changes that have more unique last changes to the modified files and more modified directories. Moreover, we also find that the decrease

TABLE 7. Performance ($\times 0.1$) when removing each feature. These values have subtracted the corresponding results with all features used. Negative values means positive effect, and vise versa.

Projects	Metrics	Diffusion				Size			Purpose	History			Experience		
		NS	ND	NF	Entropy	LA	LD	LT	FIX	NDEV	AGE	NUC	EXP	REXP	SEXP
BUG	Recall	0.041	0.083	-0.030	-0.462	-0.030	0.012	-0.018	-0.030	-0.030	-0.095	-0.008	-0.024	-0.002	0.000
	Precision	0.092	0.001	0.030	0.070	0.076	0.016	-0.006	-0.190	0.061	-0.572	0.000	-0.057	0.000	0.003
	F1-score	0.058	0.059	-0.013	-0.336	0.003	0.010	-0.015	-0.079	-0.004	-0.248	-0.058	-0.036	-0.014	0.000
COL	Recall	-0.029	0.007	0.007	-0.463	0.051	-0.015	-0.250	-0.088	-0.169	-0.066	-1.199	-0.051	0.044	-0.096
	Precision	0.024	-0.096	-0.046	0.053	-0.094	-0.099	0.087	0.043	0.058	-0.027	-0.594	-0.048	0.087	-0.038
	F1-score	0.005	-0.059	-0.027	-0.156	-0.042	-0.070	-0.046	-0.007	-0.031	-0.043	-0.847	-0.051	0.072	-0.061
JDT	Recall	0.073	-0.018	0.075	0.053	0.337	0.043	-0.035	-0.902	-0.165	-0.096	-0.215	0.002	-0.136	-0.285
	Precision	-0.036	-0.139	-0.038	-0.173	-0.132	0.004	0.012	0.198	0.013	0.011	-0.145	-0.032	0.025	0.125
	F1-score	-0.023	-0.142	-0.025	-0.166	-0.079	0.011	0.006	-0.013	-0.018	-0.006	-0.182	-0.031	-0.003	0.063
MOZ	Recall	0.018	0.070	0.088	-0.056	-0.025	-0.043	-0.132	-0.352	-0.010	-0.016	-0.156	-0.138	-0.103	-0.161
	Precision	-0.009	-0.034	-0.075	-0.146	-0.011	-0.005	-0.138	0.026	-0.063	-0.005	-0.211	0.022	0.018	0.018
	F1-score	-0.011	-0.038	-0.090	-0.194	-0.017	-0.009	-0.189	0.000	-0.081	-0.008	-0.287	0.016	0.014	0.009
PLA	Recall	0.095	0.319	0.151	0.680	0.306	-0.033	0.200	-2.439	-0.024	-0.042	0.646	0.143	0.112	-0.187
	Precision	-0.067	-0.234	-0.068	-0.281	-0.146	0.037	-0.085	0.438	0.011	0.035	-0.308	-0.114	-0.024	0.047
	F1-score	-0.403	-0.166	-0.033	-0.156	-0.080	0.027	-0.040	-0.452	0.005	0.024	-0.189	-0.079	0.000	0.003
POS	Recall	-0.159	0.100	0.023	-0.258	-0.039	-0.008	-0.041	-0.112	-0.098	-0.119	-0.395	-0.059	-0.090	-0.119
	Precision	-0.063	-0.130	-0.051	-0.141	-0.030	-0.002	-0.004	0.044	0.053	-0.077	-0.095	-0.037	-0.030	0.016
	F1-score	-0.102	-0.050	-0.025	-0.014	-0.034	-0.004	-0.018	-0.015	-0.004	-0.094	-0.214	-0.046	-0.054	-0.036
AVE	Recall	0.007	0.093	0.052	-0.084	0.100	-0.007	-0.046	-0.653	-0.083	-0.072	-0.233	-0.021	-0.032	-0.142
	Precision	-0.010	-0.105	-0.041	-0.056	-0.056	-0.008	-0.022	0.093	0.022	-0.106	-0.225	-0.044	0.013	0.028
	F1-score	-0.019	-0.066	-0.035	-0.170	-0.041	-0.006	-0.050	-0.094	-0.022	-0.063	-0.296	-0.038	0.003	-0.004

TABLE 8. Comparison between our NF model with previous state-of-the-art models for traditional within-project defect prediction. The best result for each dataset under each metric is highlighted in **red**.

Data	EALR		LT		OW		CBS		NF	
	F1	P_{opt}	F1	P_{opt}	F1	P_{opt}	F1	P_{opt}	F1	P_{opt}
BUG	0.537	0.723	0.338	0.730	0.408	0.752	0.580	0.731	**0.649**	**0.956**
COL	0.468	0.599	0.344	0.838	0.387	0.852	0.548	0.618	**0.608**	**0.930**
JDT	0.251	0.494	0.191	0.798	0.191	0.798	0.365	0.650	**0.423**	**0.922**
MOZ	0.100	0.463	0.061	0.658	0.061	0.658	0.220	0.624	**0.289**	**0.951**
PLA	0.256	0.551	0.185	0.759	0.185	0.759	0.378	0.709	**0.432**	**0.944**
POS	0.401	0.523	0.274	0.792	0.274	0.792	0.551	0.627	**0.584**	**0.944**
AVE	0.336	0.559	0.232	0.763	0.251	0.769	0.440	0.660	**0.498**	**0.941**

of F1-score is the most when we remove the features of Entropy, NUC, FIX, ND, and AGE. It demonstrates that Entropy, NUC, FIX, ND, and AGE are the major features for the performance of our prediction model, which is consistent with our findings in Section VI-A.

C. HOW WELL DOES THE NF MODEL PERFORM WHEN APPLIED TO TRADITIONAL WITHIN-PROJECT JUST-IN-TIME DEFECT PREDICTION?

To demonstrate the generalization of the proposed NF model, we further apply it to traditional within-project defect prediction. Table 8 displays the evaluation results. Here, we also only report the results of F1-score and P_{opt} metrics since F1-score is the weighted harmonic average of the recall and precision.

From Table 8, we can find that when the proposed NF is applied to traditional defect prediction, it still achieves better results than baselines. NF performs much better than EALR, LT, OneWay, and CBS. All of the highlights lie at the column of NF with no exception. The average F1-score values of NF are 16.2%, 26.6%, 24.7%, and 5.8% higher than EALR, LT, OneWay, and CBS, respectively. Hence we can conclude that NF achieves state-of-the-art performance on the metric of F1-score for traditional within-project JIT-SDP. For the metric of P_{opt}, NF achieves significantly higher results than other competitors, *e.g.*, on average, $\sim$38%, $\sim$18%, $\sim$17% and $\sim$28% higher than EALR, LT, OneWay and CBS, respectively. The average P_{opt} value of NF is 94.1%. Hence it is clear that the proposed NF performs better than others in many applications.

VII. THREATS TO VALIDITY

Several threats may have an effect on our results. This section presents the threats to the internal and external validity.

Internal validity: The threat to internal validity mainly come from feature extraction. Our model starts from hand-crafted features extracted by previous studies, and the quality of these features, of course, will influent the prediction performance of our model. However, there have been many studies focusing on exploring various features in the context of defect-inducing changes [7], [34], [71], [72]. We have carefully selected features that can well reflect change properties to mitigate this threat. Maybe new powerful features that can better describe the changes can further improve our model.

External validity: Although we utilize six open source datasets to evaluate our model, these datasets might not be representative for all software projects, *e.g.*, commercial projects. However, Kamei *et al.* [9] have conducted study on these six open source projects and five commercial projects, and the experimental results on these open source projects and additional commercial projects are similar. Hence we believe our study in this paper is also general.

VIII. CONCLUSIONS AND FUTURE WORK

Many software engineering tasks heavily rely on the classification/regression of hand-crafted features. Previous literature usually adopts manually designed feature selection techniques to handle the feature correlations and intercon-

nections. In this paper, we propose the NF model that unifies the advantages of two different worlds: neural networks and decision forests. The neural networks can automatically learn high dimensional feature representations, and decision forests perform well on high dimensional data problems. We manage to combine them in a holistic system that can be trained in an end-to-end manner. We apply our model to defect prediction (JIT-SDP) to demonstrate its superiority in representation. With hand-crafted features as input, our model achieves significantly better results than previous state-of-the-art methods. It demonstrates that our model can learn high-quality feature representations from hand-crafted features. In the future, we plan to adopt our NF model to other software engineering tasks for feature classification or regression, such as defect prediction, software code review, software vulnerability discovery, software requirements, and malware detection.

REFERENCES

[1] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in Int. Conf. Pred. Mod. Data Anal. Softw. Eng. (PROMISE), 2007, p. 9.
[2] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller, "Predicting faults from cached history," in Int. Conf. Softw. Eng. (ICSE), 2007, pp. 489–498.
[3] 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. (TSE), vol. 34, no. 4, pp. 485–496, 2008.
[4] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in Int. Conf. Softw. Eng. (ICSE), 2016, pp. 297–308.
[5] E. A. Felix and S. P. Lee, "Integrated approach to software defect prediction," IEEE Access, vol. 5, pp. 21 524–21 547, 2017.
[6] Z. Xu, S. Li, Y. Tang, X. Luo, T. Zhang, J. Liu, and J. Xu, "Cross version defect prediction with representative data via sparse subset selection," in Int. Conf. Prog. Compreh. (ICPC), 2018, pp. 132–143.
[7] A. Mockus and D. M. Weiss, "Predicting risk of software changes," Bell Labs Tech. J., vol. 5, no. 2, pp. 169–180, 2000.
[8] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang, "An industrial study on the risk of software changes," in ACM SIGSOFT Symp. Found. Softw. Eng. (FSE), 2012, p. 62.
[9] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," IEEE Trans. Softw. Eng. (TSE), vol. 39, no. 6, pp. 757–773, 2013.
[10] P. Tourani and B. Adams, "The impact of human discussions on just-in-time quality assurance: An empirical study on openstack and eclipse," in Int. Conf. Softw. Anal. Evol. Reeng. (SANER), vol. 1, 2016, pp. 189–200.
[11] R. Malhotra and M. Khanna, "An empirical study for software change prediction using imbalanced data," Empir. Softw. Eng. (ESE), vol. 22, no. 6, pp. 2806–2851, 2017.
[12] Y. Guo, M. Shepperd, and N. Li, "Bridging effort-aware prediction and strong classification: a just-in-time software defect prediction study," in Int. Conf. Softw. Eng. (ICSE), 2018, pp. 325–326.
[13] X. Yu, J. Liu, Z. Yang, X. Jia, Q. Ling, and S. Ye, "Learning from imbalanced data for predicting the number of software defects," in Int. Symp. Softw. Reliab. Eng. (ISSRE), 2017, pp. 78–89.
[14] J. Liu, Y. Zhou, Y. Yang, H. Lu, and B. Xu, "Code churn: A neglected metric in effort-aware just-in-time defect prediction," in Int. Symp. Empir. Softw. Eng. Meas. (ESEM), 2017, pp. 11–19.
[15] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, and A. E. Hassan, "Studying just-in-time defect prediction using cross-project models," Empir. Softw. Eng. (ESE), vol. 21, no. 5, pp. 2072–2106, 2016.
[16] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, and H. Leung, "Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models," in ACM SIGSOFT Symp. Found. Softw. Eng. (FSE), 2016, pp. 157–168.
[17] Q. Huang, X. Xia, and D. Lo, "Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction," in Int. Conf. Softw. Maint. Evol. (ICSME), 2017, pp. 159–170.
[18] W. Fu and T. Menzies, "Revisiting unsupervised learning for defect prediction," in ACM SIGSOFT Symp. Found. Softw. Eng. (FSE), 2017, pp. 72–83.
[19] X. Yang, D. Lo, X. Xia, and J. Sun, "TLEL: A two-layer ensemble learning approach for just-in-time defect prediction," Inform. Softw. Tech. (IST), vol. 87, pp. 206–220, 2017.
[20] X. Chen, Y. Zhao, Q. Wang, and Z. Yuan, "Multi: Multi-objective effort-aware just-in-time software defect prediction," Inform. Softw. Tech. (IST), vol. 93, pp. 1–13, 2018.
[21] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in Int. Conf. Softw. Qual. Reliab. Secur. (QRS), 2015, pp. 17–26.
[22] L. Zhao, Z. Shang, L. Zhao, A. Qin, and Y. Y. Tang, "Siamese dense neural network for software defect prediction with small data," IEEE Access, vol. 7, pp. 7663–7677, 2018.
[23] S. Sukhbaatar, J. Weston, R. Fergus et al., "End-to-end memory networks," in Adv. Neural Inform. Process. Syst. (NIPS), 2015, pp. 2440–2448.
[24] D. Bahdanau, J. Chorowski, D. Serdyuk, P. Brakel, and Y. Bengio, "End-to-end attention-based large vocabulary speech recognition," in IEEE Int. Conf. Acoust. Spee. SP (ICASSP), 2016, pp. 4945–4949.
[25] K. Wang, B. Babenko, and S. Belongie, "End-to-end scene text recognition," in Int. Conf. Comput. Vis. (ICCV), 2011, pp. 1457–1464.
[26] M. Riedmiller and H. Braun, "A direct adaptive method for faster backpropagation learning: The RPROP algorithm," in IEEE Int. Conf. Neural Networks, 1993, pp. 586–591.
[27] Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, "Efficient backprop," in Neural Networks: Tricks of the Trade, 1998, pp. 9–50.
[28] L. Breiman, "Random forests," Mach. Learn., vol. 45, no. 1, pp. 5–32, 2001.
[29] R. Caruana, N. Karampatziakis, and A. Yessenalina, "An empirical evaluation of supervised learning in high dimensions," in Int. Conf. Mach. Learn. (ICML), 2008, pp. 96–103.
[30] P. Kontschieder, M. Fiterau, A. Criminisi, and S. R. Bulo, "Deep neural decision forests," in Int. Conf. Comput. Vis. (ICCV), 2015, pp. 1467–1475.
[31] R. Rahman, S. Haider, S. Ghosh, and R. Pal, "Design of probabilistic random forests with applications to anticancer drug sensitivity prediction," Cancer Inform., vol. 14, pp. CIN–S30 794, 2015.
[32] A. Dapogny and K. Bailly, "Face alignment with cascaded semi-parametric deep greedy neural forests," Pattern Recogn. Let. (PRL), vol. 102, pp. 75–81, 2018.
[33] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in Int. Conf. Softw. Eng. (ICSE), 2006, pp. 452–461.
[34] A. E. Hassan, "Predicting faults using the complexity of code changes," in Int. Conf. Softw. Eng. (ICSE), 2009, pp. 78–88.
[35] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener, "Defect prediction from static code features: current results, limitations, new approaches," Automat. Softw. Eng. (ASE), vol. 17, no. 4, pp. 375–407, 2010.
[36] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with noise in defect prediction," in Int. Conf. Softw. Eng. (ICSE), 2011, pp. 481–490.
[37] E. Giger, M. D'Ambros, M. Pinzger, and H. C. Gall, "Method-level bug prediction," in Int. Symp. Empir. Softw. Eng. Meas. (ESEM), 2012, pp. 171–180.
[38] F. Thung, X.-B. D. Le, and D. Lo, "Active semi-supervised defect categorization," in Int. Conf. Prog. Compreh. (ICPC), 2015, pp. 60–70.
[39] Y. Kamei and E. Shihab, "Defect prediction: Accomplishments and future challenges," in Int. Conf. Softw. Anal. Evol. Reeng. (SANER), vol. 5, 2016, pp. 33–45.
[40] Y. Liu, Y. Li, J. Guo, Y. Zhou, and B. Xu, "Connecting software metrics across versions to predict defects," in Int. Conf. Softw. Anal. Evol. Reeng. (SANER), 2018, pp. 232–243.
[41] J. Chen, Y. Yang, K. Hu, Q. Xuan, Y. Liu, and C. Yang, "Multiview transfer learning for software defect prediction," IEEE Access, vol. 7, pp. 8901–8916, 2019.
[42] S. Kim, E. J. Whitehead Jr, and Y. Zhang, "Classifying software changes: Clean or buggy?" IEEE Trans. Softw. Eng. (TSE), vol. 34, no. 2, pp. 181–196, 2008.
[43] S. McIntosh and Y. Kamei, "Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction," IEEE Trans. Softw. Eng. (TSE), vol. 44, no. 5, pp. 412–428, 2018.

[44] T. Mende and R. Koschke, "Revisiting the evaluation of defect prediction models," in Int. Conf. Pred. Mod. Data Anal. Softw. Eng. (PROMISE), 2009, p. 7.

[45] T. Mende and R. Koschke, "Effort-aware defect prediction models," in Eur. Conf. Softw. Maint. Reeng. (CSMR), 2010, pp. 107–116.

[46] L. Pascarella, F. Palomba, and A. Bacchelli, "Fine-grained just-in-time defect prediction," J. Syst. Softw. (JSS), 2018.

[47] T. Fukushima, Y. Kamei, S. McIntosh, K. Yamashita, and N. Ubayashi, "An empirical study of just-in-time defect prediction using cross-project models," in Work. Conf. Min. Softw. Repos. (MSR), 2014, pp. 172–181.

[48] Y. Li, Z. Huang, Y. Wang, and B. Fang, "Evaluating data filter on cross-project defect prediction: Comparison and improvements," IEEE Access, vol. 5, pp. 25 646–25 656, 2017.

[49] Z. Xu, P. Yuan, T. Zhang, Y. Tang, S. Li, and Z. Xia, "HDA: cross-project defect prediction via heterogeneous domain adaptation with dictionary learning," IEEE Access, vol. 6, pp. 57 597–57 613, 2018.

[50] B. Ghotra, S. McIntosh, and A. E. Hassan, "Revisiting the impact of classification techniques on the performance of defect prediction models," in Int. Conf. Softw. Eng. (ICSE), 2015, pp. 789–800.

[51] D. Bowes, T. Hall, and J. Petrić, "Software defect prediction: do different classifiers find the same defects?" Softw. Qual. J. (SQJ), vol. 26, no. 2, pp. 525–552, 2018.

[52] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," in Int. Conf. Softw. Eng. (ICSE), 2015, pp. 99–108.

[53] D. Ryu, J.-I. Jang, and J. Baik, "A transfer cost-sensitive boosting approach for cross-project defect prediction," Softw. Qual. J. (SQJ), vol. 25, no. 1, pp. 235–272, 2017.

[54] Q. Song, Y. Guo, and M. Shepperd, "A comprehensive investigation of the role of imbalanced learning for software defect prediction," IEEE Trans. Softw. Eng. (TSE), 2018.

[55] K. E. Bennin, J. Keung, P. Phannachitta, A. Monden, and S. Mensah, "Mahakil: Diversity based oversampling approach to alleviate the class imbalance issue in software defect prediction," IEEE Trans. Softw. Eng. (TSE), vol. 44, no. 6, pp. 534–550, 2018.

[56] Z. Li, X.-Y. Jing, F. Wu, X. Zhu, B. Xu, and S. Ying, "Cost-sensitive transfer kernel canonical correlation analysis for heterogeneous defect prediction," Automat. Softw. Eng. (ASE), vol. 25, no. 2, pp. 201–245, 2018.

[57] S. Huda, K. Liu, M. Abdelrazek, A. Ibrahim, S. Alyahya, H. Al-Dossari, and S. Ahmad, "An ensemble oversampling model for class imbalance problem in software defect prediction," IEEE Access, vol. 6, pp. 24 184–24 195, 2018.

[58] H. Liu and R. Setiono, "Chi2: Feature selection and discretization of numeric attributes," in IEEE Int. Conf. Tools Artif. Intell. (ICTAI), 1995, pp. 388–391.

[59] M. Dash and H. Liu, "Feature selection for classification," Intell. Data Anal., vol. 1, no. 1-4, pp. 131–156, 1997.

[60] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," J. Mach. Learn. Res. (JMLR), vol. 3, no. Mar, pp. 1157–1182, 2003.

[61] T. M. Cover and J. A. Thomas, Elements of information theory. John Wiley & Sons, 2012.

[62] J. Li, K. Cheng, S. Wang, F. Morstatter, R. P. Trevino, J. Tang, and H. Liu, "Feature selection: A data perspective," ACM Comput. Surv., vol. 50, no. 6, p. 94, 2018.

[63] H. Uğuz, "A two-stage feature selection method for text categorization by using information gain, principal component analysis and genetic algorithm," Knowl.-Based Syst., vol. 24, no. 7, pp. 1024–1032, 2011.

[64] M. A. Hall, "Correlation-based feature selection for discrete and numeric class machine learning," in Int. Conf. Mach. Learn. (ICML), 2000, pp. 359–366.

[65] R. Li, W. Zhong, and L. Zhu, "Feature screening via distance correlation learning," J. Am. Stat. Assoc., vol. 107, no. 499, pp. 1129–1139, 2012.

[66] S. Huda, S. Alyahya, M. M. Ali, S. Ahmad, J. Abawajy, H. Al-Dossari, and J. Yearwood, "A framework for software defect prediction and metric selection," IEEE Access, vol. 6, pp. 2844–2858, 2017.

[67] Q. Yu, J. Qian, S. Jiang, Z. Wu, and G. Zhang, "An empirical study on the effectiveness of feature selection for cross-project defect prediction," IEEE Access, 2019.

[68] I. H. Laradji, M. Alshayeb, and L. Ghouti, "Software defect prediction using ensemble learning on selected features," Inform. Softw. Tech. (IST), vol. 58, pp. 388–402, 2015.

[69] J. Jiarpakdee, C. Tantithamthavorn, and A. E. Hassan, "The impact of correlated metrics on defect models," IEEE Trans. Softw. Eng. (TSE), 2018.

[70] J. Jiarpakdee, C. Tantithamthavorn, and C. Treude, "Autospearman: Automatically mitigating correlated software metrics for interpreting defect models," in Int. Conf. Softw. Maint. Evol. (ICSME), 2018, pp. 92–103.

[71] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in Work. Conf. Min. Softw. Repos. (MSR), 2010, pp. 31–41.

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

[73] A. G. Koru, D. Zhang, K. El Emam, and H. Liu, "An investigation into the functional form of the size-defect relationship for software modules," IEEE Trans. Softw. Eng. (TSE), vol. 35, no. 2, pp. 293–304, 2009.

[74] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, "Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows," in Int. Conf. Softw. Eng. (ICSE), 2010, pp. 495–504.

[75] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram, "How do fixes become bugs?" in ACM SIGSOFT Symp. Found. Softw. Eng. (FSE). ACM, 2011, pp. 26–36.

[76] S. Matsumoto, Y. Kamei, A. Monden, K.-i. Matsumoto, and M. Nakamura, "An analysis of developer metrics for fault prediction," in Int. Conf. Pred. Mod. Data Anal. Softw. Eng. (PROMISE), 2010, p. 18.

[77] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," IEEE Trans. Softw. Eng. (TSE), vol. 26, no. 7, pp. 653–661, 2000.

[78] M. M. Öztürk, "Which type of metrics are useful to deal with class imbalance in software defect prediction?" Inform. Softw. Tech. (IST), vol. 92, pp. 17–29, 2017.

[79] S. Hosseini, B. Turhan, and M. Mäntylä, "A benchmark study on the effectiveness of search-based data selection and feature selection for cross project defect prediction," Inform. Softw. Tech. (IST), vol. 95, pp. 296–312, 2018.

[80] Z. Xu, J. Liu, X. Luo, Z. Yang, Y. Zhang, P. Yuan, Y. Tang, and T. Zhang, "Software defect prediction based on kernel pca and weighted extreme learning machine," Inform. Softw. Tech. (IST), vol. 106, pp. 182–200, 2019.

[81] L. Breiman, Classification and regression trees. Chapman & Hall/CRC, 1984.

[82] J. R. Quinlan, "Induction of decision trees," Mach. Learn., vol. 1, no. 1, pp. 81–106, 1986.

[83] J. R. Quinlan, C4.5: Programs for machine learning. Morgan Kaufmann, 1993.

[84] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in Int. Conf. Mach. Learn. (ICML), 2010, pp. 807–814.

[85] B. C. Csáji, "Approximation with artificial neural networks," Fac. Sci. Etvs Lornd Univ. Hungary, vol. 24, p. 48, 2001.

[86] S. Rota Bulo and P. Kontschieder, "Neural decision forests for semantic image labelling," in IEEE Conf. Comput. Vis. Pattern Recog. (CVPR), 2014, pp. 81–88.

[87] M. J. Kochenderfer, Decision making under uncertainty: theory and application. MIT press, 2015.

[88] L. Rokach and O. Maimon, "Top-down induction of decision trees classifiers-a survey," IEEE Trans. Syst. Man Cy. C., vol. 35, no. 4, pp. 476–487, 2005.

[89] D. N. Reshef, Y. A. Reshef, H. K. Finucane, S. R. Grossman, G. McVean, P. J. Turnbaugh, E. S. Lander, M. Mitzenmacher, and P. C. Sabeti, "Detecting novel associations in large data sets," Science, vol. 334, no. 6062, pp. 1518–1524, 2011.

[90] T. Jiang, L. Tan, and S. Kim, "Personalized defect prediction," in Int. Conf. Automat. Softw. Eng. (ASE), 2013, pp. 279–289.

[91] F. Provost and T. Fawcett, "Robust classification for imprecise environments," Mach. Learn., vol. 42, no. 3, pp. 203–231, 2001.

[92] T. Fawcett, "An introduction to roc analysis," Pattern Recogn. Let. (PRL), vol. 27, no. 8, pp. 861–874, 2006.

[93] Q. Song, Z. Jia, M. Shepperd, S. Ying, and J. Liu, "A general software defect-proneness prediction framework," IEEE Trans. Softw. Eng. (TSE), vol. 37, no. 3, pp. 356–370, 2011.

[94] S. Shivaji, E. J. Whitehead, R. Akella, and S. Kim, "Reducing features to improve code change-based bug prediction," IEEE Trans. Softw. Eng. (TSE), vol. 39, no. 4, pp. 552–569, 2013.

[95] X. Xia, D. Lo, S. McIntosh, E. Shihab, and A. E. Hassan, "Cross-project build co-change prediction," in Int. Conf. Softw. Anal. Evol. Reeng. (SANER), 2015, pp. 311–320.

[96] N. Cliff, Ordinal methods for behavioral data analysis. Psychology Press, 2014.
[97] Y. Benjamini and Y. Hochberg, "Controlling the false discovery rate: a practical and powerful approach to multiple testing," J. R. Stat. Soc. B, vol. 57, no. 1, pp. 289–300, 1995.
[98] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek, "Appropriate statistics for ordinal level data: Should we really be using t-test and cohenâĂŹsd for evaluating group differences on the nsse and other surveys," in Annu. Meet. Florida Assoc. Instit. Res., 2006, pp. 1–33.
[99] E. Arisholm, L. C. Briand, and E. B. Johannessen, "A systematic and comprehensive investigation of methods to build and evaluate fault prediction models," J. Syst. Softw. (JSS), vol. 83, no. 1, pp. 2–17, 2010.