IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 18 April 2025, accepted 30 April 2025, date of publication 5 May 2025, date of current version 16 May 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3567140

RESEARCH ARTICLE

Understanding Software Defect Prediction Through eXplainable Neural Additive Models

RUIQI HE, YONG LI, AND CHI SUN
College of Computer Science and Technology, Xinjiang Normal University, Ürümqi, Xinjiang 830054, China
Corresponding author: Yong Li (liyong@live.com)

This work was supported in part by Xinjiang Key Research and Development Program under Grant 2022B01007-1, and in part by the Natural Science Foundation of Xinjiang Uygur Autonomous Region under Grant 2022D01A225.

ABSTRACT Software defect prediction, leveraging machine learning techniques to proactively identify potential defects in software systems, plays a crucial role in enhancing software quality and reliability. However, a major challenge in this field lies in the opacity of the prediction process and the lack of interpretability of the results, which significantly limits its practical application. To address this issue, this paper introduces eXplainable Neural Additive Models (XNAMs). The proposed model constructs single-feature inputs for software defect data, enabling transparent visualization of the impact of individual features on prediction outcomes. Additionally, it employs feature gradient analysis to examine the average absolute values of feature gradients during forward propagation, thereby quantifying and comparing the contribution of each feature to the decision-making process. Furthermore, feature interaction analysis is conducted to uncover nonlinear interactions between different features. Experimental evaluations on six software projects demonstrate that XNAMs outperform existing models in prediction performance while offering clear explanations of feature contributions, ensuring high transparency and practical applicability.

INDEX TERMS Software defect prediction, explainable neural additive models, feature gradients, interpretability.

I. INTRODUCTION

Effective prediction and management of software defects have become critical for ensuring software quality and reliability, as software failures during operation can lead to significant economic losses and, in some cases, even fatal consequences. Software defect prediction, by identifying potential failure-inducing defects at an early stage, not only reduces the cost of remediation and improves software reliability but also significantly enhances user satisfaction [1]. The field has now evolved towards Just-in-Time (JIT) defect prediction, which analyzes a developer's submission history to predict, in real time, whether a newly submitted code change is likely to introduce defects. This allows rapid intervention before defects propagate, making JIT defect prediction a major research focus in software defect prediction [2].

With the rapid advancement of artificial intelligence, various complex models have been widely applied in software defect prediction, achieving remarkable performance. Despite their superior predictive capabilities, traditional deep learning models suffer from a lack of transparency in their decision-making processes, making it difficult for developers to understand and trust their outputs. This issue is particularly critical when precisely identifying and fixing specific software defects [2]. As a result, model interpretability has become a key research direction in machine learning. In domains such as medical diagnosis [3], autonomous driving [4], criminal justice [5], and financial risk assessment [6], both accurate prediction and transparent decision-making are equally crucial. Given the highly specialized nature of software defect prediction, there is an

The associate editor coordinating the review of this manuscript and approving it for publication was Porfirio Tramontana.

urgent need for interpretable models to support software development and maintenance decisions [31].

To address these challenges, this study introduces eXplainable Neural Additive Models (XNAMs). Through experimental validation on six different software projects, we demonstrate the advantages of XNAMs in improving defect prediction accuracy while intuitively illustrating the contribution of each feature to the prediction outcome. This approach not only enhances developers' trust in predictive models but also opens new possibilities for software quality assessment and requirements analysis in broader software engineering applications. The adoption of XNAMs signifies a promising direction for integrating explainable machine learning techniques into software engineering, paving the way for broader applicability and trustworthiness in AI-driven defect prediction. This research not only advances software defect prediction technology but also provides a novel perspective for the study of explainable machine learning in this field.

To summarize, this work is motivated by the critical need to enhance the interpretability of software defect prediction models without sacrificing predictive accuracy. Accordingly, we highlight the following key contributions of this study:

- We propose **XNAMs (eXplainable Neural Additive Models)**, a novel architecture that integrates the modeling flexibility of neural networks with the inherent interpretability of additive models.
- We design a **three-level interpretability mechanism**, comprising gradient-based feature importance, individual feature response analysis, and feature interaction visualization, enabling both global and local explanation of model behavior.
- We perform **extensive experiments** on six real-world software projects, demonstrating that XNAMs achieve superior predictive performance compared to state-of-the-art baselines while ensuring high transparency.
- We provide **actionable insights** for developers and software engineers, showing how interpretable outputs can support informed decision-making in defect localization, code review, and software quality management.

These contributions collectively establish XNAMs as a promising and practical solution for bridging the gap between model performance and interpretability in software defect prediction.

The remainder of this paper is organized as follows. Section II reviews the related work on explainable artificial intelligence and software defect prediction. Section III introduces the architecture and interpretability mechanisms of the proposed XNAMs model. Section IV describes the experimental setup, including datasets, model configurations, and evaluation metrics. Section V presents and analyzes the experimental results. Section VI compares XNAMs with state-of-the-art models. Section VII explores the interpretability analysis of model behavior. Finally, Section VIII concludes the paper and outlines directions for future work.

II. RELATED WORK

A. EXPLAINABLE ARTIFICIAL INTELLIGENCE

Explainable Artificial Intelligence (XAI) has undergone rapid development as a tool for addressing real-world problems. It has been widely adopted in handling complex data tasks such as image classification, text classification, audio classification, and time-series forecasting, owing to its superior performance. However, these models are often regarded as ''non-interpretable models'', meaning that their decision-making processes lack transparency [30]. As non-interpretable models are increasingly deployed in critical environments to make high-stakes predictions, there is a growing demand for transparency from various stakeholders in the AI community. The inherent risk lies in making and using irrational, unlawful decisions or failing to provide detailed explanations of their behavior [31].To address this issue, various methods have been developed to enhance model interpretability. For example, Locally Interpretable Model-Agnostic Explanations (LIME) [22] and Shapley values [23] provide tools for explaining individual predictions of complex models, improving transparency by quantifying each feature's contribution to the model's decision. Additionally, sensitivity-based approaches, although limited to simple network architectures, attempt to identify key factors influencing predictions based on statistical significance [24].

Model interpretability can be classified into intrinsic interpretability (pre-hoc interpretability) and post-hoc interpretability [25]. Intrinsic interpretability refers to models that are inherently designed with explainable characteristics, meaning they do not require additional interpretability tools. Since these models are constructed with interpretability in mind, their explanations are built-in prior to training and deployment, making them pre-hoc interpretable models, also known as transparent models.In contrast, post-hoc interpretability refers to explanations derived after model training and deployment, utilizing external tools and methods to analyze the model's behavior. These post-hoc techniques can be model-specific, such as tree-based approaches [26] and convolutional neural network-based methods [27], or model-agnostic, applicable to a variety of models.

Currently, research in explainable machine learning predominantly focuses on model-agnostic post-hoc interpretability methods, which often struggle to provide high-fidelity and consistent explanations.In comparison, self-explainable models remain relatively underexplored in the field of explainable machine learning. These models often face the fundamental challenge of balancing interpretability and predictive performance [30], making it a crucial area for further research and development.

B. SOFTWARE DEFECT PREDICTION

Early approaches to software defect prediction relied on statistical analysis and expert systems, which often lacked the capability to handle complex data. With the introduction of machine learning techniques, such as logistic regression [7],

Bayesian classifiers [8], support vector machines (SVMs) [7], artificial neural networks (ANNs) [9], decision tree classifiers [10], random forest algorithms [11], kernel principal component analysis (KPCA) [12], deep learning [13], and ensemble learning techniques [14], the accuracy of software defect prediction has significantly improved.

In the field of Just-in-Time (JIT) software defect prediction, deep learning techniques have been widely adopted due to their strong feature extraction capabilities. These models are particularly effective in capturing complex patterns in code submissions, but their non-interpretable model nature remains a significant challenge.Despite the substantial research effort devoted to improving prediction accuracy in software defect prediction, the lack of interpretability in model decision-making remains an underexplored issue. Studies have shown that software defect prediction models often lack transparency, making it difficult to explain their decisions. This leads to reduced trust among developers and decision-makers in real-world software development environments [15], [16], [17]. The lack of interpretability in software analytics makes it difficult for developers to understand the basis of predictions, ultimately undermining the credibility of these models in practice [18], [19], [20], [21]. Providing interpretability in defect prediction models is as important and beneficial as improving their accuracy. However, existing research has primarily focused on enhancing predictive performance, neglecting efforts to make file-level defect prediction models more interpretable [16]. As a result, the interpretability of defect prediction models remains largely unexplored, and the lack of explainability significantly restricts the practical adoption of software defect prediction techniques in software development workflows.

A review of the literature indicates that maintaining predictive performance while improving model interpretability is an urgent challenge. To address this issue, this study proposes an eXplainable Neural Additive Model (XNAMs), which provides a comprehensive explanation of the software defect prediction process through three core interpretability modules.During model construction, an individual input tensor is created for each feature in the dataset. This feature isolation mechanism enables the model to accurately quantify the marginal effect of a single feature, thereby revealing the independent impact of key indicators such as code modification volume (nf) and developer experience (sexp) on defect prediction. Furthermore, TensorFlow's GradientTape API is utilized to monitor specific inputs. By analyzing the mean absolute value of feature gradients during forward propagation, the model quantifies the contribution of each feature to decision-making.Additionally, a feature interaction analysis module is introduced, which constructs a two-dimensional response surface to systematically evaluate nonlinear interaction effects between features. This in-depth interaction analysis provides important insights for team collaboration and code review processes, offering valuable references for optimizing software development workflows.

III. EXPLAINABLE NEURAL ADDITIVE MODELS (XNAMs)

To address the issue of lack of interpretability in software defect prediction, this study proposes an eXplainable Neural Additive Model (XNAMs) to improve model transparency and interpretability.

Generalized Additive Models (GAMs) are a type of regression model that allows nonlinear relationships to be represented through additive functions [28]. The mathematical form of GAMs is given by:

$$g(E[y]) = \beta_0 + f_1(x_1) + f_2(x_2) + \cdots + f_n(x_n). \quad (1)$$

where $E[y]$ represents the expected value of the response variable y, $g(\cdot)$ is the link function, f_i denotes the smoothing function for each feature, which can be nonlinear, x_i represents the feature variables, and β_0 is the intercept term of the model. The advantage of Generalized Additive Models (GAMs) lies in their interpretability: the effect of each feature is modeled independently, allowing for a clear and intuitive understanding of each feature's contribution to the response variable.

Neural Additive Models (NAMs) process each input feature through an independent neural network [29], combining the interpretability advantages of Generalized Additive Models (GAMs) with the high-performance learning capabilities of neural networks. This architecture allows NAMs to maintain both flexibility and accuracy while ensuring model interpretability. Each feature-specific network can be customized based on the characteristics of the data and the requirements of the prediction task, enhancing the model's ability to capture complex data patterns and optimizing overall network performance.

Building upon GAMs and NAMs, eXplainable Neural Additive Models (XNAMs) further enhance interpretability by introducing three key feature explanation functionalities: individual feature importance, gradient-based feature importance, and feature interaction analysis. Gradient-based feature importance quantifies the impact of each feature by computing the gradient of the model's output with respect to the input features. Specifically, TensorFlow's GradientTape API is used to monitor specified inputs and analyze the mean absolute value of feature gradients during the forward propagation process. A larger gradient magnitude indicates a more significant contribution of the feature to the model's decision. By leveraging this approach, XNAMs explicitly assign an influence score to each feature, thereby quantifying its contribution to the final prediction outcome. Independent feature output analysis is implemented by constructing input tensors that retain only the raw values of the current feature while setting all other feature values to zero. This allows for an evaluation of each feature's independent effect on the prediction. Feature interaction analysis systematically evaluates and visualizes nonlinear interactions between features by constructing a uniform grid over the feature space and forming high-dimensional response surfaces. This module first selects the most influential feature subset based on gradient importance, then computes an interaction strength

matrix to quantify feature dependencies. This approach not only uncovers complex interdependencies among features but also provides quantitative insights for designing differentiated software quality control strategies. Figure 1 presents the overall framework of XNAMs.

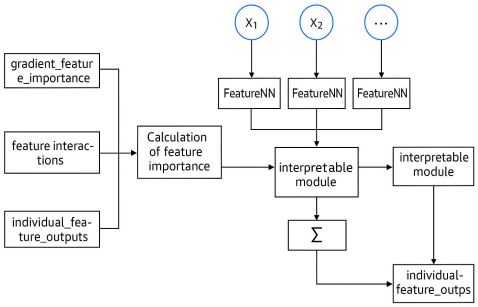

FIGURE 1. Overview of the proposed XNAMs framework.

In the fundamental architecture of XNAMs, each input feature is processed by an independent neural network, referred to as FeatureNN. Each FeatureNN takes a single feature as input, and the network structure of each feature is completely independent, allowing the model to evaluate and interpret each feature's contribution separately. Each FeatureNN consists of one or more hidden layers, which utilize the ReLU activation function to capture nonlinear relationships between features and prediction outcomes. The outputs from all FeatureNNs are then aggregated through a weighted summation, yielding the final model prediction. This summation process effectively integrates the individual contributions of each feature into the overall prediction. The number of hidden layers and neurons per layer in each FeatureNN is adjustable. In the baseline model, each FeatureNN typically contains two hidden layers, with 64 neurons in the first layer and 32 neurons in the second layer. This design is intended to capture nonlinear relationships between features, making it well-suited for tasks with moderate complexity. For more complex tasks, users can increase the number of hidden layers or adjust the number of neurons per layer to better capture intricate patterns. For example, a deeper network architecture might include three or more hidden layers, with 128 neurons in the first layer, 64 neurons in the second layer, and 32 neurons in the third layer, designed to model more sophisticated feature interactions. For simpler tasks, a shallower network architecture may be more effective. In a shallow network, each FeatureNN might consist of only a single hidden layer with 64 neurons, which is sufficient for cases where feature relationships are relatively simple and do not require multiple layers of complex transformations.

Through this architecture, XNAMs achieve a balance between flexibility and efficient nonlinear modeling, while ensuring model interpretability. The gradient-based feature importance module enables a clear quantification of each feature's contribution, providing greater transparency and accuracy for practical applications. This design offers a new direction for explainability in deep learning models, particularly in high-risk domains, where interpretability is crucial. Furthermore, it opens up new possibilities for applying explainable machine learning techniques in fields such as software engineering.

A. GRADIENT-BASED FEATURE IMPORTANCE

In the field of software defect prediction, computing feature importance is crucial for understanding how the model identifies potential defects. This study employs a gradient-based approach, which quantifies feature importance by computing the gradient of the model's prediction output with respect to the input features. The importance score of feature i, denoted as SI_i, is defined as:

$$SI_i = \frac{\frac{1}{N}\sum_{j=1}^{N}\left|\frac{\partial y_j}{\partial x_{ij}}\right|}{\sum_{i=1}^{M}\frac{1}{N}\sum_{j=1}^{N}\left|\frac{\partial y_j}{\partial x_{ij}}\right|} \tag{2}$$

where N is the number of samples, M is the number of features, and x_{ij} represents the value of the i-th feature in the j-th sample. The term $\frac{\partial y_j}{\partial x_{ij}}$ denotes the sensitivity of the model output with respect to the feature x_i, measuring how changes in x_i influence the model's predictions. Algorithm 1 presents the pseudocode for this method.

This method evaluates the local sensitivity of each feature to the model's prediction, providing an intuitive way to identify which features have the most significant impact on the prediction outcomes. By leveraging this approach, software engineers gain a transparent and interpretable tool to identify and rectify the code components most likely to introduce defects. Consequently, this enhances code quality and helps reduce future maintenance costs.

B. INDEPENDENT FEATURE RESPONSE

To gain a deeper understanding of the specific impact of each feature on the model's output, this study proposes a method for computing *Independent Feature Response*. This approach analyzes the marginal effect of a single feature on the model's prediction while keeping all other features fixed at their mean values. The response function for feature k, denoted as $R_k(x)$, is defined as:

$$R_k(x) = f\left([\overline{x_1}, \ldots, \overline{x_{k-1}}, x, \overline{x_{k+1}}, \ldots, \overline{x_M}]\right) \tag{3}$$

where x represents the value of feature k, $\overline{x_i}$ denotes the mean value of feature i, and M is the total number of features.

To obtain a complete response curve, we uniformly sample n points within the feature value range $[\min_k, \max_k]$:

$$x_i = \min_k + \frac{i}{n-1}(\max_k - \min_k), \quad i = 0, 1, \ldots, n-1. \tag{4}$$

This method reveals the nonlinear relationship between the feature and the prediction outcome.

This approach enables the quantification of each feature's independent impact, clarifying its contribution to the

Algorithm 1 Gradient-Based Feature Importance Method

Require: Model f, Dataset X, Batch size b
Ensure: Feature importance scores S

Initialize feature importance scores: importance_scores $= \mathbf{0}$ (zero vector of length num_features)
Calculate the total number of batches: num_batches $= \lceil |X|/b \rceil$
for each batch i from 1 to num_batches **do**
    Get the current batch data: $X_{\text{batch}} = \text{get\_batch}(X, i, b)$
    Compute the gradient of the current batch's prediction with respect to input features:

$$\text{grads} = \frac{\partial f(X_{\text{batch}})}{\partial X_{\text{batch}}}$$

    Compute the average of the absolute gradients for the current batch:

$$\text{batch\_importance} = \text{mean}(|\text{grads}|, \text{axis} = 0)$$

    Accumulate the feature importance scores for the current batch:

$$\text{importance\_scores} += \text{batch\_importance}$$

end for
Normalize feature importance scores:

$$\text{importance\_scores} /= \text{num\_batches}$$

$$\text{importance\_scores} /= \sum \text{importance\_scores}$$

return importance_scores

model's output while keeping all other features constant. Although complex interactions often exist among features, the Independent Feature Response method provides a concise way to assess the relative importance of individual features. This helps identify the most influential features, making it particularly suitable for software defect prediction. It can reveal which code attributes—such as code complexity or modification frequency—contribute most to defect prediction. Algorithm 2 presents the pseudocode for this method.

By leveraging the Independent Feature Response method, the independent contribution of each feature within the model can be effectively assessed. This provides software engineers with an intuitive decision-support tool, aiding in the optimization of model prediction accuracy and the enhancement of software quality.In practical applications, response curve-based analysis facilitates the identification and adjustment of key features that significantly impact model performance, thereby improving the stability and security of software systems.

C. FEATURE EXCHANGE EFFECT

Feature interaction effect analysis focuses on the joint influence of multiple features to identify potential interaction patterns between them. Traditional feature importance analysis often overlooks the interdependencies among features, whereas feature interaction effect analysis provides a more granular explanation of the model by deeply exploring the dependencies between feature pairs. For a feature pair (i, j), the interaction effect matrix $IM_{ij}(x_i, x_j)$ is computed as follows:

$$IM_{ij}(x_i, x_j) = f\left([\overline{x_1}, \ldots, \overline{x_{k-1}}, x, \overline{x_{k+1}}, \ldots, \overline{x_M}]\right) - \left[R_i(x_i) + R_j(x_j) - f(\overline{x})\right] \quad (5)$$

where (x_i, x_j) represents a combination of feature values, $\overline{x}$ denotes the mean vector of all features, and $R_i(x_i)$ and $R_j(x_j)$ are the independent responses of features i and j, respectively. To enhance computational efficiency, the analysis is conducted only for highly important features, and grid sampling is applied to compute the interaction effects:

$$x_i^p = \min_i + \frac{p}{n-1}(\max_i - \min_i), \quad p = 0, 1, \ldots, n-1. \quad (6)$$

$$x_j^q = \min_j + \frac{q}{n-1}(\max_j - \min_j), \quad q = 0, 1, \ldots, n-1. \quad (7)$$

Algorithm 3 presents the pseudocode for this method.

This method effectively captures joint effects between features, helping to reveal complex dependencies among them. These interaction effects provide deeper insights into software defect prediction, thereby enhancing the interpretability of predictive models.In practical applications, feature interaction effect analysis offers valuable guidance for software engineers, enabling them to identify critical feature interactions during development. By leveraging these insights, engineers can proactively minimize the likelihood of defect occurrences, ultimately improving software quality and reliability.

Algorithm 2 Individual Feature Response Analysis

Require: Model f, Dataset X, Number of sampling points n
Ensure: Feature response curves R

Calculate the mean of all features: $x_{\text{mean}} = \text{mean}(X, \text{axis} = 0)$
Get the number of features: num_features $= X$.shape[1]
Initialize the response dictionary: responses $= \{\}$
for each feature i from 1 to num_features **do**
    Get the range of the current feature:

$$x_{\min} = \min(X[:, i]), \quad x_{\max} = \max(X[:, i])$$

    Generate equidistant sampling points:

$$x_{\text{points}} = \text{linspace}(x_{\min}, x_{\max}, n)$$

    Create input data, fix other features to the mean:

$$X_{\text{input}} = \text{repeat}(x_{\text{mean}}, n).\text{reshape}(n, \text{num\_features})$$

    Assign sampled values to the current feature:

$$X_{\text{input}}[:, i] = x_{\text{points}}$$

    Calculate the model's predictions:

$$y_{\text{pred}} = f(X_{\text{input}})$$

    Store the current feature's response curve in the dictionary:

$$\text{responses}[i] = \{\text{'x\_values'} : x_{\text{points}}, \text{'y\_values'} : y_{\text{pred}}\}$$

end for
return all feature response curves: return responses

IV. EXPERIMENTAL DESIGN

A. DATASETS

To ensure the breadth and depth of this study, six open-source projects were selected as datasets: QT, OpenStack, Eclipse JDT, Eclipse Platform, Gerrit, and Go. These projects are influential in the open-source community and encompass different programming languages, including C, Java, and Go, enhancing the generalizability of the research findings. The widely-used SZZ algorithm was applied to identify and analyze defect-inducing commits. This process involved analyzing commit messages to identify defect-fixing commits, using the git diff command to detect the lines of code changed in these commits, and finally using git blame to determine the commits that introduced the defects. To ensure data accuracy and relevance, blank, comment, and merge commits were excluded, resulting in a dataset of six projects with 310,370 commits, including 81,300 commits marked as defect-related.

Metadata about code changes plays a crucial role in just-in-time software defect prediction, and a rich set of features was extracted from these changes to improve the model's performance and interpretability. Based on existing research, these features were categorized into nine main dimensions: change size, code distribution, change purpose, developer experience, code complexity, textual content, code structure, file modification history, and code review. The change size dimension reflects the extent of code modifications. Studies have shown that larger code changes are more likely to introduce defects. The number of lines added or deleted is used to measure change size, and relative line counts are used to reduce the potential correlation between these metrics. The code distribution dimension represents how widely the code change is spread across different files. Code changes that affect multiple files require developers to have a broader understanding of the codebase, which may increase the risk of introducing defects. Metrics such as the number of files, directories, and subsystems affected by the change are used to quantify code distribution, and the entropy of the changes has been shown to be an effective predictor of defects.

The change purpose dimension reveals the intent of the code commit. In general, defect-fixing changes tend to be more complex and more likely to introduce new defects.

TABLE 1. Data information.

Project	#Changes	%Defect	Language
QT	95758	15.16	C++
OpenStack	66065	31.68	C++
JDT	13348	41.20	Java
Platform	39365	37.74	Java
Gerrit	34610	8.64	Java
Go	61224	36.75	Golang

Algorithm 3 Feature Interaction Analysis

Require: Model f, Dataset X, Top k Features, Grid size n
Ensure: Interaction effect matrix I

Get the feature importance scores:

$$\text{importance\_scores} = \text{gradient\_feature\_importance}(f, X)$$

Get the top k features based on importance ranking:

$$\text{top\_features} = \text{argsort}(\text{importance\_scores})[-k:]$$

Calculate the mean of all features:

$$x_{\text{mean}} = \text{mean}(X, \text{axis} = 0)$$

Initialize the interaction effect dictionary:

$$\text{interaction\_effects} = \{\}$$

for each feature pair (i, j) from top_features **do**
    Get the range of the current features:

$$x_{i_{\min}}, x_{i_{\max}} = \min(X[:, i]), \max(X[:, i])$$
$$x_{j_{\min}}, x_{j_{\max}} = \min(X[:, j]), \max(X[:, j])$$

    Generate grid points:

$$x_i = \text{linspace}(x_{i_{\min}}, x_{i_{\max}}, n)$$
$$x_j = \text{linspace}(x_{j_{\min}}, x_{j_{\max}}, n)$$
$$X_i, X_j = \text{meshgrid}(x_i, x_j)$$

    Initialize the interaction matrix I:

$$I = \text{zeros}(n, n)$$

    for each grid point $p = 1$ to n **do**
        for each grid point $q = 1$ to n **do**
            Copy the mean feature vector:

$$x_{\text{input}} = x_{\text{mean}}.\text{copy}()$$

            Set feature values at the current grid point:

$$x_{\text{input}}[i] = X_i[p, q], \quad x_{\text{input}}[j] = X_j[p, q]$$

            Compute the model prediction:

$$I[p, q] = f(x_{\text{input}})$$

        end for
    end for
    Store the current feature pair's interaction effects in the dictionary:

$$\text{interaction\_effects}[(i, j)] = \{\text{'feature1\_values'} : x_i, \text{'feature2\_values'} : x_j, \text{'interaction\_matrix'} : I\}$$

end for
return all feature interaction effects:

$$\text{return interaction\_effects}$$

This dimension is measured by analyzing whether the change is a defect fix and the number of defect reports related to the change. The developer experience dimension focuses on measuring the experience level of the developer. Research has shown that developer experience is directly related to software quality. To quantify developer experience, factors such as the frequency of past commits, recent activity, and experience with the subsystem affected by the change are considered. Through a detailed analysis of these dimensions, prediction models can more accurately identify potential defect risks, allowing for more effective defect management strategies during the development process. Our dataset

TABLE 2. Summary of changed data characteristics.

Dimension	Feature Name	Characterization
Size	LA/LD	More lines of code added and deleted, the change may introduce more defects.
	CA/CD	More code blocks added and deleted; the larger the impact on the software code, the more likely defects will be introduced.
	LT	The more times a file is touched, the more complex the modified code may be, increasing the likelihood of defects.
Code Distribution	NS	Changes involving more subsystems are more complex and may introduce more defects.
	ND	Changes involving more directories are more complex and may introduce more defects.
	NF	Changes involving more files are more complex and may introduce more defects.
	Entropy	The larger the entropy, the more scattered the distribution of files related to the changed code, requiring developers to understand more code, increasing the likelihood of defects.
Objective	FIX	Defect-fixing changes are more complex and are more likely to introduce new defects.
	NBR	The more related bug reports, the more code needs to be fixed, and the more likely new defects will be introduced.
	EXP	More experienced developers are less likely to introduce defects.
Approach	REXP	Developers who have recently modified code frequently or have a deeper understanding of the project are less likely to introduce defects.
	SEXP	Developers who understand the subsystems involved in the changes are less likely to introduce defects.
	Awareness	The more familiar developers are with modifying the subsystems, the less likely they are to introduce defects.

collects and organizes a detailed set of change metadata features, which are distributed across the nine key dimensions as summarized in Table 2.

B. EXPERIMENTAL SETUP

In this experiment, the eXplainable Neural Additive Model (XNAMs) was employed for training and evaluation. The model adopts an adaptive feature subnetwork structure that allows flexible configuration for both shallow and deep networks. Specifically, shallow networks are designed with a single hidden layer, where the number of neurons is determined based on the complexity of the corresponding feature. For more complex features, deep networks are configured with a three-layer decremental design, where the number of neurons in each layer decreases progressively to control model size while maintaining expressive power. This design balances flexibility and interpretability, ensuring that the model can capture complex feature behaviors without excessive computational burden.

Key hyperparameters, including learning rate, batch size, hidden layer neuron allocation, and distribution strategies, were selected through random search combined with five-fold cross-validation. To ensure computational efficiency while maintaining predictive accuracy, an initial grid search was conducted to define a refined search space. The AUC performance on the validation set was used as an early stopping criterion, and if no significant improvement was observed over multiple iterations, the tuning process was halted to avoid unnecessary computational overhead. Based on extensive experimentation, the optimal training configuration was determined as follows: 20 training epochs, an initial learning rate of 0.001 with a decay rate of 0.995 for stability, and a batch size of 32 to balance training efficiency and optimization quality.

For fair comparative evaluation, a traditional deep neural network (DNN) was implemented as the baseline model. The DNN consisted of three hidden layers with 64, 32, and 16 neurons respectively, using ReLU activation functions. The same regularization techniques applied in XNAMs were used in the baseline model to ensure a consistent experimental basis. To evaluate the impact of regularization strategies on XNAMs, an ablation study was conducted across four configurations: dropout only (dropout rate 0.2), L2 regularization only (weight decay 0.01), a combination of both techniques, and no regularization. The results indicated that the combination of dropout and L2 regularization was most effective in preventing overfitting and improving model generalization.

To further enhance robustness, a feature-level dropout mechanism was also introduced. In this strategy, entire feature subnetworks were randomly dropped during training, effectively reducing the model's reliance on any single feature and improving generalization. This multi-level regularization framework proved essential for stabilizing the model, particularly in high-dimensional settings.

In terms of feature engineering, the study utilized 14 key software metrics, including developer experience (sexp, rexp), and the number of developers (ndev). To address dimensional inconsistencies across features, all metrics were standardized prior to training. The model's ability to capture

individual feature effects and their interactions was evaluated using both feature importance analysis and interaction analysis across multiple datasets.

Finally, a set of comprehensive evaluation metrics was employed to assess model performance, including AUROC (Area Under the Receiver Operating Characteristic Curve), precision, recall, and F1-score. These metrics provide a holistic view of both the predictive capability and practical applicability of XNAMs. Through careful hyperparameter tuning and ablation study design, this experimental setup ensures that XNAMs achieve strong predictive performance while maintaining interpretability across diverse software defect prediction tasks.

V. RESULTS

A. ANALYSIS OF RESULTS

Our model's performance evaluation across different software projects demonstrates its superior capability in predicting software defects. Table 3 presents the performance evaluation results of the model across various projects.

When analyzing the performance evaluation results of the XNAMs model across different software projects, it can be observed that the model demonstrates consistently strong performance across all datasets.

Specifically, on the QT dataset, the model achieves an AUC of 0.7600, indicating good classification capability in distinguishing defective and non-defective code submissions. On the JDT dataset, the AUC is 0.7575, slightly lower than that of QT but still demonstrating strong predictive ability, suggesting that XNAMs can effectively handle diverse project datasets. For OpenStack and Platform datasets, the AUC values are 0.7645 and 0.7789, respectively, further validating the cross-project adaptability of the model. Notably, on the Platform dataset, the model achieves a high AUC score, indicating that XNAMs maintain stable predictive performance even in complex datasets with diverse features. A particularly noteworthy result is observed in the Gerrit dataset, where XNAMs achieve the highest AUC of 0.8157 among all tested datasets. This reflects the model's exceptional adaptability in code review systems, demonstrating its ability to accurately capture key defect prediction features. However, on the Go dataset, the AUC is 0.7425, which is relatively lower than other datasets. Despite this, XNAMs still maintain strong competitiveness compared to existing methods. This indicates that XNAMs exhibit generalization capability across different project datasets, though there is room for improvement in certain cases, such as the Go project. From the perspective of precision, recall, and F1-score, XNAMs exhibit strong stability and adaptability across different datasets. The balanced improvement in precision and recall enhances the model's overall performance. Specifically, in the Gerrit dataset, both precision and recall achieve high performance, further supporting the model's superior adaptability in specialized domains. Even in the Go project, despite the lower AUC, the model maintains relatively high precision and recall, indicating robust stability across multiple scenarios.

Table 4 presents the results of the ablation study, demonstrating that L2 regularization significantly enhances the model's generalization ability, particularly in high-dimensional datasets, where it effectively mitigates overfitting.Meanwhile, Dropout also plays a crucial role in reducing overfitting. However, an excessively high Dropout rate may lead to information loss, thereby negatively impacting model performance.Ultimately, the combination of L2 regularization and Dropout achieves the best performance, leveraging the strengths of both techniques to improve the stability and predictive accuracy of the model. Through this ablation study, we further validate the importance of regularization techniques in XNAMs and provide practical insights for model optimization in real-world applications. When applying XNAMs to high-dimensional datasets, computational costs and memory consumption increase significantly. Since XNAMs train an independent neural network for each input feature, an increase in feature dimensionality directly leads to a surge in computational resource demands, exacerbating training time and memory usage, particularly when storing large numbers of model parameters and intermediate computations. Furthermore, the expansion of feature dimensions introduces greater optimization complexity, especially when complex interactions exist between high-dimensional features. Gradient updates in such cases can lead to computational bottlenecks, affecting both model convergence speed and stability. Additionally, challenges such as gradient vanishing or gradient explosion in high-dimensional feature interactions may further impede model learning effectiveness. To mitigate these challenges, regularization techniques (e.g., Dropout and L2 regularization) are commonly employed to reduce overfitting. However, these techniques also introduce additional computational burdens and require precise hyperparameter tuning. To enhance the scalability and performance of XNAMs, dimensionality reduction, feature selection, or parameter sharing strategies serve as effective solutions. Moreover, optimizing regularization techniques is crucial for balancing model performance and computational efficiency.

VI. COMPARISON WITH OTHER MODELS

Table 5 presents a comparison of AUC results between XNAMs and four other software defect prediction models (DeepJIT, CC2Vec, LR-JIT, and DBN-JIT). The results indicate that XNAMs consistently outperform competing models across multiple datasets.

On the QT dataset, XNAMs achieve an AUC of 0.7600, significantly higher than DeepJIT (0.7144), CC2Vec (0.7164), LR-JIT (0.6843), and DBN-JIT (0.6858). This demonstrates that XNAMs can more accurately capture feature information in the QT dataset, further showcasing their adaptability. Similarly, the results on the OpenStack dataset highlight the superior performance of XNAMs, with an AUC of 0.7645, whereas DeepJIT and CC2Vec achieve

TABLE 3. Performance of XNAMs across different datasets.

Project	QT	JDT	OpenStack	Platform	Gerrit	Go
AUC	0.7600	0.7575	0.7645	0.7789	0.8157	0.7425
Precision	0.6836	0.6823	0.6876	0.6995	0.7399	0.6883
Recall	0.7079	0.6659	0.7050	0.7154	0.7386	0.6544
F1-score	0.6955	0.6740	0.6962	0.7073	0.7393	0.6709

TABLE 4. AUC performance across different projects in ablation study.

Project	QT	JDT	OpenStack	Platform	Gerrit	Go
Baseline	0.7380	0.7255	0.7480	0.7665	0.8032	0.7289
L2 Regularization Only	0.7450	0.7340	0.7515	0.7720	0.8082	0.7325
Dropout Only	0.7500	0.7405	0.7550	0.7740	0.8105	0.7358
L2 + Dropout	0.7600	0.7575	0.7645	0.7789	0.8157	0.7425

only 0.7140 and 0.7078, respectively—over 5 percentage points lower than XNAMs. Additionally, the traditional models LR-JIT and DBN-JIT exhibit even lower AUC values (0.6750 and 0.6627, respectively), further emphasizing the substantial performance improvement achieved by XNAMs in this scenario. For the JDT dataset, XNAMs achieve an AUC of 0.7575, slightly outperforming DeepJIT (0.7491) and CC2Vec (0.7466), while maintaining a clear advantage over LR-JIT and DBN-JIT. This suggests that even in scenarios where XNAMs compete with deep learning models, they retain strong predictive performance, effectively integrating feature interpretability with prediction accuracy. On the Platform dataset, XNAMs demonstrate a significant advantage, achieving an AUC of 0.7789, whereas DeepJIT and CC2Vec score only 0.6912, with LR-JIT and DBN-JIT performing even worse (0.6912 and 0.6781, respectively). The superior performance of XNAMs on this dataset highlights their ability to handle complex feature interactions, leading to a notable improvement in predictive accuracy. For the Gerrit dataset, XNAMs once again achieve the highest AUC of 0.8157, surpassing DeepJIT (0.7875) and CC2Vec (0.7873). While LR-JIT (0.8131) achieves a score close to XNAMs, the interpretability of XNAMs makes them a more practical and insightful choice. Finally, on the Go dataset, XNAMs achieve an AUC of 0.7425, outperforming DeepJIT (0.7314) and CC2Vec (0.7244), while showing a substantial advantage over LR-JIT (0.6783) and DBN-JIT (0.6805). This indicates that XNAMs maintain strong predictive performance, even in datasets where other models struggle.

The study also employed statistical analysis methods to evaluate the performance and robustness of the eXplainable Neural Additive Model (XNAMs) in software defect prediction tasks. The experimental results were systematically assessed using multiple quantitative metrics, revealing the model's stability and effectiveness across different tasks. In terms of predictive performance, XNAMs demonstrated outstanding results on the Qt dataset, with ROC curve analysis showing an AUC value of 0.76, significantly outperforming traditional machine learning methods. Further analysis indicated that the model achieved a good balance between precision (0.68) and recall (0.70), resulting in an F1-score of 0.69, which highlights the model's stable discriminative capability in defect identification. To assess the robustness of the model, cross-validation was conducted, and the performance standard deviation across different data splits (± 0.03) confirmed its strong generalization ability. Notably, the model exhibited highly consistent predictive performance when handling code modifications of varying scales, which is of significant practical importance. In the feature stability analysis, the coefficient of variation of feature importance was calculated. The results showed that key features, such as the number of developers and modification entropy, maintained a stable ranking across different experiments, further reinforcing the model's reliability in real-world applications. These statistical findings indicate that XNAMs not only achieve excellent predictive performance in software defect prediction but also exhibit strong robustness and interpretability. These multidimensional advantages provide strong support for the model's practical application in software development.

A. ANALYSIS OF INTERPRETABLE RESULTS

In the explainability research of software defect prediction, this study systematically explores the explainability features of XNAMs through systematic experimental design and a multidimensional analytical framework. The research analyzes three key dimensions: feature importance, feature response patterns, and feature interaction effects, leading to a series of findings with significant theoretical and practical implications.

Taking the OpenStack dataset as an example, we demonstrate the explainability performance of NAMs in software defect prediction tasks, revealing several valuable insights. In terms of feature importance, Figure 2 shows that the number of files (nf) is the most influential predictor, with an importance score of 0.7585. This is followed by the number of fixes (`fix`) and file age (`age`), with importance scores of 0.0657 and 0.0523, respectively. The substantial disparity in importance scores suggests that human resource allocation plays a crucial role in software quality assurance. Notably, traditional metrics, such as lines of code changed (`la`) and

TABLE 5. Comparison of XNAMs and other model AUC results.

Model	QT	OpenStack	JDT	Platform	Gerrit	Go
DeepJIT	0.7144	0.7140	0.7491	0.6912	0.7875	0.7314
CC2Vec	0.7164	0.7078	0.7466	0.6912	0.7873	0.7244
LR-JIT	0.6843	0.6750	0.7497	0.6910	0.8131	0.6783
DBN-JIT	0.6858	0.6627	0.7267	0.6781	0.7757	0.6805
XNAMs	0.7600	0.7575	0.7645	0.7789	0.8157	0.7425

file history (`exp_avg`), exhibit relatively lower predictive power in this study.

Methodologically, this divergence can be attributed to XNAMs' use of nonlinear feature response modeling, which enables the capture of complex, non-monotonic relationships between features and defect risk. This contrasts with the predominantly linear correlation-based analyses used in prior studies. As a result, certain features such as file age, which were previously considered marginal, demonstrate more nuanced and impactful behaviors in our framework. These findings not only corroborate parts of the existing literature but also introduce new perspectives on the dynamics of feature contribution in software defect prediction.

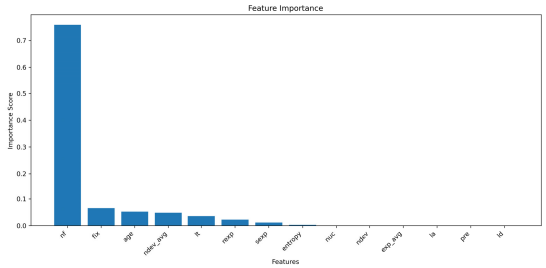

FIGURE 2. Gradient-based feature importance scores on the OpenStack project.

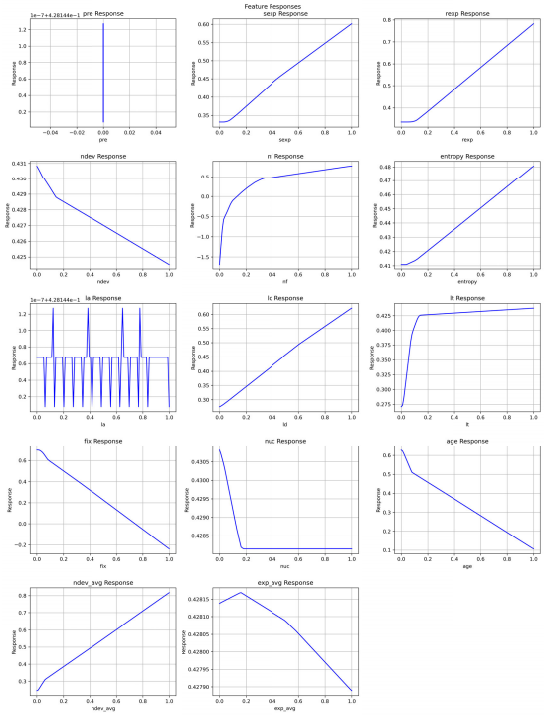

FIGURE 3. Independent feature response on the OpenStack project.

Feature response analysis reveals the nonlinear relationships between predictive factors and the target variable. By analyzing the response curves of individual features, the study identifies complex nonlinear patterns in several key features. As shown in Figure 3, the number of code modifications (NF) exhibits a positive correlation with defect probability, but the trend flattens in the high-value range. This suggests that beyond a certain threshold, further increases in code modifications have a diminished impact on defect risk. These findings provide important theoretical support for resource allocation and risk management in the software development process.

In the feature interaction analysis, as shown in Figure 4, several significant interaction effects were observed. Most notably, there was a strong correlation between the number of developers and modification entropy, suggesting that the synergy between team size and code complexity could exponentially increase defect risk. Additionally, a significant interaction pattern was observed between code change size and modification entropy, highlighting the potential risk associated with large-scale code changes in high-complexity scenarios.

Based on these empirical findings, several management insights can be drawn: First, software project managers should carefully control the scale of concurrent development activities, particularly when handling highly complex modules. Second, it is essential to establish stricter code review mechanisms, especially for large-scale refactoring efforts. Finally, it may be beneficial to reduce reliance on file history features and instead focus more attention on the dynamic characteristics of current development activities. These findings not only deepen our understanding of the mechanisms behind software defect formation, but also provide practical guidance for improving software development practices.

In the field of software defect prediction, the effectiveness of feature selection techniques and their impact on model performance have long been central research questions. Experimental results indicate significant differences in contributions among various feature categories. These differences are not only reflected in individual feature importance scores but also manifest in feature interaction effects. Code complexity-related features, such as entropy

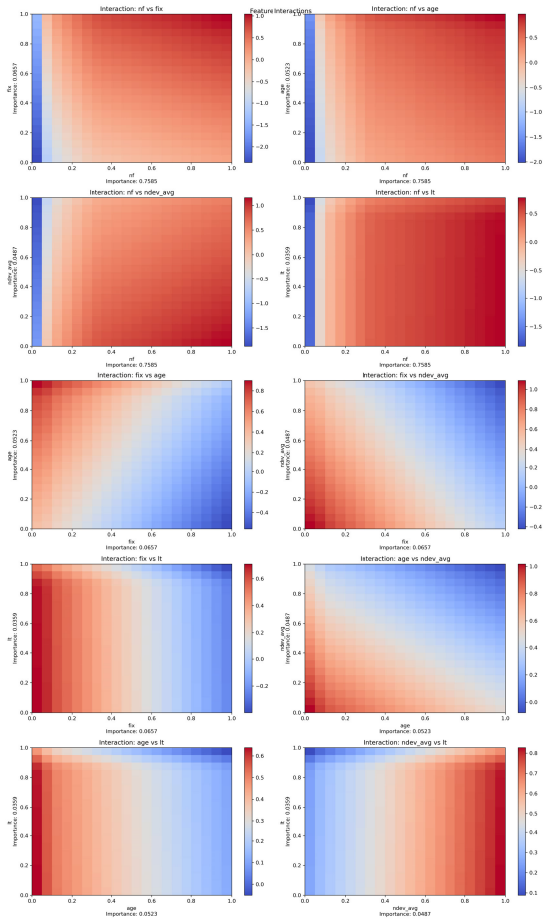

FIGURE 4. Feature interaction response on the OpenStack project.

and the number of developers, make substantial contributions to defect prediction. This finding aligns with experience from the software engineering domain and provides quantitative guidance for feature selection. Feature interaction effect analysis reveals significant synergies between certain features, highlighting that single-feature importance evaluations cannot fully capture the actual contributions of features. Thus, both independent importance and interaction effects should be considered in feature selection processes, which has important practical implications for optimizing feature selection strategies.

In summary, this study systematically validates the significant impact of feature selection techniques on model performance and offers an actionable feature evaluation framework. These findings have important theoretical and practical implications for the field of software defect prediction. From a theoretical perspective, the research confirms that XNAMs can effectively capture and quantify complex relationships between software metrics and defect risks, introducing a new paradigm for explainable machine learning in software engineering. Practically, the multidimensional interpretability information provided by the model supports concrete decision-making in software quality assurance, including resource optimization based on feature importance, practice improvements guided by response curves, and team collaboration enhancements informed by interaction effects.

Regarding the impact of class imbalance handling techniques on feature importance, experimental results show that random oversampling has significant and complex effects on feature sensitivity. Balancing the data set altered the distribution of feature importance. Developer-related features, particularly the number of developers (`ndev`) and experience metrics (`exp_avg`), exhibited higher importance on balanced data sets, reflecting the model's ability to more comprehensively capture the influence of human resource factors on defect formation. Meanwhile, code complexity-related features such as modification entropy (`entropy`) and lines of code changed (`la`) demonstrated relatively stable importance, suggesting that their contributions to the prediction results are strongly independent and less influenced by class distribution. In the feature interaction analysis, balancing the classes strengthened nonlinear relationships between features. In particular, the interaction effect between the number of developers and modification entropy was significantly amplified, showing that the model can identify more nuanced feature combination patterns on balanced data. Changes in feature response curves further confirmed this observation, as the model became more sensitive to boundary values of feature ranges on balanced datasets.

VII. CONCLUSION

This study introduces the XNAMs (eXplainable Neural Additive Models), a model characterized by excellent transparency and interpretability. Its application in just-in-time software defect prediction highlights dual advantages: enhanced predictive performance and improved model explainability. Experimental results demonstrate that XNAMs deliver superior predictive performance across multiple open-source projects, notably achieving significantly higher AUC scores than traditional methods on the Openstack and Platform datasets. Additionally, XNAMs effectively explain how individual features influence predictions, thereby increasing the model's transparency and practicality. Ablation experiments on regularization techniques confirm the effectiveness of L2 regularization and Dropout in improving model stability and generalization. Regarding the handling of correlated features and redundancy in feature-specific neural networks, experimental findings reveal both notable advantages and challenges. XNAMs' independent feature learning network architecture excels at capturing the nonlinear impact of individual features. However, when strong correlations exist between features (e.g., number of developers and code change volume), this approach can lead to information redundancy and wasted computational resources. To address this issue, a feature dropout mechanism was introduced, randomly dropping certain feature networks to reduce redundant learning and improve computational efficiency.

In terms of feature network redundancy, experiments show that some feature networks learn highly similar patterns, particularly when dealing with strongly correlated software metrics. To mitigate this, the study proposed two strategies: (1) applying regularization constraints to limit network complexity, and (2) incorporating an attention mechanism to dynamically adjust the contribution weights of individual feature networks. These approaches successfully reduce redundant computation, enhance efficiency, and maintain the model's interpretability.

Despite these positive results, XNAMs still have certain limitations. The model faces challenges with highly correlated features and exhibits high computational overhead when handling extremely large datasets. Moreover, the computational complexity of analyzing feature interaction effects increases as the number of features grows, limiting its scalability in high-dimensional settings. Future research can focus on optimizing these methods, exploring multi-feature joint response analysis frameworks, and further improving the model's applicability and computational efficiency.

XNAMs provide a new methodological approach to software defect prediction, achieving significant improvements not only in accuracy but also in model explainability. By enhancing interpretability, XNAMs offer robust support for software development and quality control. With the continued advancement of interpretable machine learning technologies, XNAMs hold promise for broader applications in software engineering, delivering more transparent and reliable decision support to the industry.

VIII. EXPERIMENTAL ENVIRONMENT

The experiments were conducted in a Python programming environment, using TensorFlow 1.15.0 as the deep learning framework to ensure backward compatibility. The experimental setup ran on Windows 11 Professional, equipped with a 2.4 GHz Intel i7-13620H processor and 16 GB of memory.

REFERENCES

[1] G. Li-Na, J. Shu-Juan, and J. Li, "Research progress of software defect prediction," *J. Softw.*, vol. 30, no. 10, pp. 3090–3114, Mar. 2019.

[2] C. Liang, F. Yuan-Rui, and Y. Meng, "Just-in-time software defect prediction: Literature review," *J. Softw.*, vol. 30, no. 5, pp. 1288–1307, 2019.

[3] R. Miotto, F. Wang, S. Wang, X. Jiang, and J. T. Dudley, "Deep learning for healthcare: Review, opportunities and challenges," *Briefings Bioinf.*, vol. 19, no. 6, pp. 1236–1246, Nov. 2018.

[4] J. Choi, T. Kim, D. Ryu, J. Baik, and S. Kim, "Just-in-time defect prediction for self-driving software via a deep learning model," *J. Web Eng.*, vol. 22, no. 2, pp. 303–326, Jun. 2023.

[5] J. Zeng, B. Ustun, and C. Rudin, "Interpretable classification models for recidivism prediction," *J. Roy. Stat. Soc. A, Statist. Soc.*, vol. 180, no. 3, pp. 689–722, Jun. 2017.

[6] D. Brigo, X. Huang, A. Pallavicini, and H. S. de Ocariz Borde, "Interpretability in deep learning for finance: A case study for the Heston model," 2021, *arXiv:2104.09476*.

[7] C. Shan, B. Chen, C. Hu, J. Xue, and N. Li, "Software defect prediction model based on LLE and SVM," in *Proc. Commun. Secur. Conf. (CSC)*, May 2014, pp. 1–5.

[8] A. Rahim, Z. Hayat, and M. Abbas, "Software defect prediction with naïve Bayes classifier," in *Proc. Int. Bhurban Conf. Appl. Sci. Technol. (IBCAST)*, Mar. 2021, pp. 293–297.

[9] M. A. Khan, N. S. Elmitwally, S. Abbas, S. Aftab, M. Ahmad, M. Fayaz, and F. Khan, "Software defect prediction using artificial neural networks: A systematic literature review," *Sci. Program.*, vol. 2022, May 2022, Art. no. 2117339.

[10] P. D. Singh and A. Chug, "Software defect prediction analysis using machine learning algorithms," in *Proc. 7th Int. Conf. Cloud Comput., Data Sci. Eng.-Confluence*, Jan. 2017, pp. 775–781.

[11] Y. N. Soe, P. I. Santosa, and R. Hartanto, "Software defect prediction using random forest algorithm," in *Proc. 12th South East Asian Tech. Univ. Consortium (SEATUC)*, vol. 1, Mar. 2018, pp. 1–5.

[12] 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," *Inf. Softw. Technol.*, vol. 106, pp. 182–200, Feb. 2019.

[13] W. Wu, C. Feng, H. Ren, X. Han, and X. Tong, "Research on software defect prediction system based on deep learning," *Proc. SPIE*, vol. 12500, pp. 1455–1462, Dec. 2022.

[14] I. H. Laradji, M. Alshayeb, and L. Ghouti, "Software defect prediction using ensemble learning on selected features," *Inf. Softw. Technol.*, vol. 58, pp. 388–402, Feb. 2015.

[15] H. K. Dam, T. Tran, and A. Ghose, "Explainable software analytics," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng., New Ideas Emerg. Technol. Results (ICSE-NIER)*, May 2018, pp. 53–56.

[16] J. Jiarpakdee, C. K. Tantithamthavorn, and J. Grundy, "Practitioners' perceptions of the goals and visual explanations of defect prediction models," in *Proc. IEEE/ACM 18th Int. Conf. Mining Softw. Repositories (MSR)*, May 2021, pp. 432–443.

[17] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead, "Does bug prediction support human developers? Findings from a Google case study," in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, May 2013, pp. 372–381.

[18] J. Jiarpakdee, C. K. Tantithamthavorn, H. K. Dam, and J. Grundy, "An empirical study of model-agnostic techniques for defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 48, no. 1, pp. 166–185, Jan. 2022.

[19] C. Khanan, W. Luewichana, K. Pruktharathikoon, J. Jiarpakdee, C. Tantithamthavorn, M. Choetkiertikul, C. Ragkhitwetsagul, and T. Sunetnanta, "JITBot: An explainable just-in-time defect prediction bot," in *Proc. 35th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Sep. 2020, pp. 1336–1339.

[20] C. K. Tantithamthavorn and J. Jiarpakdee, "Explainable AI for software engineering," in *Proc. 36th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2021, pp. 1–2.

[21] C. Tantithamthavorn, J. Jiarpakdee, and J. Grundy, "Actionable analytics: Stop telling me what it is; please tell me what to do," *IEEE Softw.*, vol. 38, no. 4, pp. 115–120, Jul. 2021.

[22] M. T. Ribeiro, S. Singh, and C. Guestrin, "'Why should I trust you?' Explaining the predictions of any classifier," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Aug. 2016, pp. 1135–1144.

[23] S. Lundberg and S. Lee, "A unified approach to interpreting model predictions," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30, Jan. 2017, pp. 1–12.

[24] K. Chen and X. Meng, "Interpretability of machine learning," *Comput. Res. Develop.*, vol. 57, no. 9, pp. 1971–1986, 2020.

[25] J.-X. Mi, A.-D. Li, and L.-F. Zhou, "Review study of interpretation methods for future interpretable machine learning," *IEEE Access*, vol. 8, pp. 191969–191985, 2020.

[26] Z. Zhou and G. Hooker, "Unbiased measurement of feature importance in tree-based methods," *ACM Trans. Knowl. Discovery Data*, vol. 15, no. 2, pp. 1–21, Apr. 2021.

[27] K. Simonyan, A. Vedaldi, and A. Zisserman, "Deep inside convolutional networks: Visualising image classification models and saliency maps," 2013, *arXiv:1312.6034*.

[28] T. J. Hastie, "Generalized additive models," in *Statistical Models in S*, vol. 2017. Evanston, IL, USA: Routledge, 2017, pp. 249–307.

[29] R. Agarwal, L. Melnick, N. Frosst, X. Zhang, B. Lengerich, R. Caruana, and G. E. Hinton, "Neural additive models: Interpretable machine learning with neural nets," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 34, Jan. 2020, pp. 4699–4711.

[30] J. Shouling, L. Jinfeng, D. Tianyu, and L. Bo, "A review of machine learning model interpretability methods, applications, and security research," *Comput. Res. Develop.*, vol. 56, no. 10, pp. 2071–2096, 2019.

[31] M. Mustaqeem, S. Mustajab, M. Alam, F. Jeribi, S. Alam, and M. Shuaib, "A trustworthy hybrid model for transparent software defect prediction: SPAM-XAI," *PLoS ONE*, vol. 19, no. 7, Jul. 2024, Art. no. e0307112.

RUIQI HE received the B.S. degree in computer science and technology from Tarim University. He is currently pursuing the degree with Xinjiang Normal University.

His research interests include interpretable machine learning and software reliability engineering.

YONG LI received the Ph.D. degree in computer science from Nanjing University of Aeronautics and Astronautics, in 2018.

He is currently a Professor with Xinjiang Normal University. His research interests include machine learning and intelligent software engineering.

CHI SUN received the bachelor's degree in software engineering from Xinjiang Normal University, where he is currently pursuing the degree.

His research interests include interpretable machine learning and intelligent software engineering.