Automatic Identification of Extrinsic Bug Reports for Just-In-Time Bug Prediction

Guisheng Fan[a,*], Yuguo Liang[a,*], Longfei Zu[a], Huiqun Yu[a,b], Zijie Huang[c], Wentao Chen[a]

[a]*Department of Computer Science and Engineering, East China University of Science and Technology, Shanghai, 200237, China.*
[b]*Shanghai Engineering Research Center of Smart Energy, Shanghai, China.*
[c]*Shanghai Key Laboratory of Computer Software Testing and Evaluating, Shanghai Development Center of Computer Software Technology, Shanghai, 201112, China*

Abstract

In software development, developers create bug reports within an Issue Tracking System (ITS) to describe the cause, symptoms, severity, and other technical details of bugs. The ITS includes reports of both intrinsic bugs (i.e., those originating within the software itself) and extrinsic bugs (i.e., those arising from third-party dependencies). Although extrinsic bugs are not recorded in the Version Control System (VCS), they can still affect Just-In-Time (JIT) bug prediction models that rely on VCS-derived information.

Previous research has shown that excluding extrinsic bugs can significantly improve JIT bug prediction model's performance. However, manually classifying intrinsic and extrinsic bugs is time-consuming and prone to errors. To address this issue, we propose a CAN model that integrates the local feature extraction capability of TextCNN with the nonlinear approximation advantage of the Kolmogorov-Arnold Network (KAN). Experiments on 1,880 labeled data samples from the OpenStack project demonstrate that the CAN model outperforms benchmark models such as BERT and CodeBERT, achieving an accuracy of 0.7492 and an F1-score of 0.8072. By comparing datasets with and without source code, we find that incorporating

*Corresponding author

Email addresses: gsfan@ecust.edu.cn (Guisheng Fan), ygliang@mail.ecust.edu.cn (Yuguo Liang), zlf@mail.ecust.edu.cn (Longfei Zu), yhq@ecust.edu.cn (Huiqun Yu), huangzj@sscenter.sh.cn (Zijie Huang), wentaochen@mail.ecust.edu.cn (Wentao Chen)

source code information enhances model performance. Finally, using the Local Interpretable Model-agnostic Explanations (LIME), an explainable artificial intelligence technique, we identify that keywords such as “test” and “api” in bug reports significantly contribute to the prediction of extrinsic bugs.

Keywords:
Bug report, Extrinsic bugs, Just-In-Time bug prediction, Explainable artificial intelligence

---

1. Introduction

Bug reports are a central artifact in software maintenance, providing crucial information for understanding, localizing, and resolving bugs. Analyzing bug reports is an important and long-standing topic in software engineering research. While traditional bug prediction research often assumes that every reported bug is linked to a corresponding bug-introducing change (BIC) in the Version Control System (VCS), recent empirical studies have revealed that this assumption does not always hold [1, 2, 3]. In particular, prior works [1, 2] point out that some failures are not directly caused by changes visible in VCS, but rather by modifications in the context or environment, such as dependency upgrades, that occur outside of the recorded code history. More explicitly, [3] distinguishes between intrinsic bugs, whose bug-introducing changes can be identified in VCS, and extrinsic bugs, which are caused by external changes (e.g., errors in external APIs, compatibility issues, or evolving specifications) and therefore lack an explicit bug-introducing change in VCS.

This distinction between intrinsic and extrinsic bugs is not merely academic. In Just-In-Time (JIT) bug prediction, a technique that predicts whether a change under review will introduce a bug before it is committed, training datasets are constructed by linking bug-fixing changes (BFCs) to their corresponding BICs. However, when extrinsic bugs are mislabeled as intrinsic, these datasets become contaminated with noisy or incorrect BICs. Such contamination has been shown to significantly degrade model performance [3], potentially leading to false alarms or missed bug-prone changes in practice. Given the increasing reliance on automated JIT bug prediction in large-scale software projects, ensuring the quality of these datasets is critical.

A straightforward solution is to manually inspect and label bug reports

to distinguish between intrinsic and extrinsic bugs, as done in [3]. However, this approach is prohibitively time-consuming, error-prone, and does not scale to the volume of bug reports generated in modern projects. Furthermore, the growing complexity of software ecosystems, characterized by rapid dependency evolution, continuous integration pipelines, and heterogeneous runtime environments, makes such manual filtering increasingly impractical. This creates an urgent need for automated methods to identify extrinsic bugs directly from bug report contents.

Despite the recognized impact of extrinsic bugs on predictive modeling, no prior work has systematically explored the use of modern text classification techniques for their automatic identification. This gap is particularly notable given the rapid progress in deep learning for software engineering, where pre-trained models of code [4, 5, 6] and natural language [7] have achieved state-of-the-art results in tasks such as bug/defect prediction [8], clone detection [9] and code summarization [10]. Additionally, prior studies on bug report quality [11, 12, 13] have shown that code snippets, stack traces, and other structured elements in bug reports can serve as strong contextual signals for understanding bugs. However, their potential in distinguishing extrinsic from intrinsic bugs remains unexplored.

To address these challenges, we propose a novel text classification approach, CAN, which integrates TextCNN [14] and KAN [15], to automatically identify extrinsic bugs from bug reports. Using datasets from the OpenStack project, we evaluate multiple representative models on both code-containing and code-absent datasets of bug reports, and employ LIME (Local Interpretable Model-agnostic Explanations) [16] to explain the predictions of the best-performing model. The contributions of this paper are summarized as follows:

- We extract bug reports from the OpenStack project and apply various text classification techniques to evaluate the effectiveness of identifying extrinsic bugs. Experimental results indicate that text classification techniques can effectively identify extrinsic bugs by analyzing bug reports. Our proposed CAN model demonstrates the best performance in terms of F1 score.

- We examine the role of source code in identifying extrinsic bugs. We hypothesize that valuable information can be extracted from the source code present in bug reports and conduct experiments to compare the

classification performance of datasets incorporating and excluding code. The experimental results indicate that datasets incorporating source code as text enhance the models' ability to identify extrinsic bugs, while excluding source code generally degrades models' performance.

- We employ LIME to analyze feature importance in the CAN model's correct predictions. Experimental results show that words such as "line" and "py" play a significant role in classifying intrinsic bugs, while terms like "test" and "api" strongly influence the classification of extrinsic bugs.

By automating the identification of extrinsic bugs, our work addresses a critical bottleneck in preparing high-quality datasets for JIT bug prediction, thereby contributing both to the theoretical understanding of bug origin classification and to the practical improvement of predictive maintenance tools.

To support reproducibility and facilitate future research, we release a replication package that includes the labeled dataset, preprocessing scripts, training and evaluation code of the CAN model. The resources are publicly available[1], allowing researchers to replicate our experiments and further explore extrinsic bug classification in different settings.

The rest of this paper is organized as follows. Section 2 presents the research questions and introduces experimental settings. Experimental results and analysis, as well as discussion and implications, are described in Section 3 and Section 4, respectively. Section 5 describes the threats to validity. Section 6 introduces related work. The conclusion and future work are presented in section 7.

2. Experimental Setup

In this section, we firstly present research questions and corresponding motivations, and then introduce the processes of data extraction, preprocessing, model construction, and performance evaluation. The overall framework of our experiment is depicted in Figure 1.

[1]https://github.com/Hugo-Liang/CAN

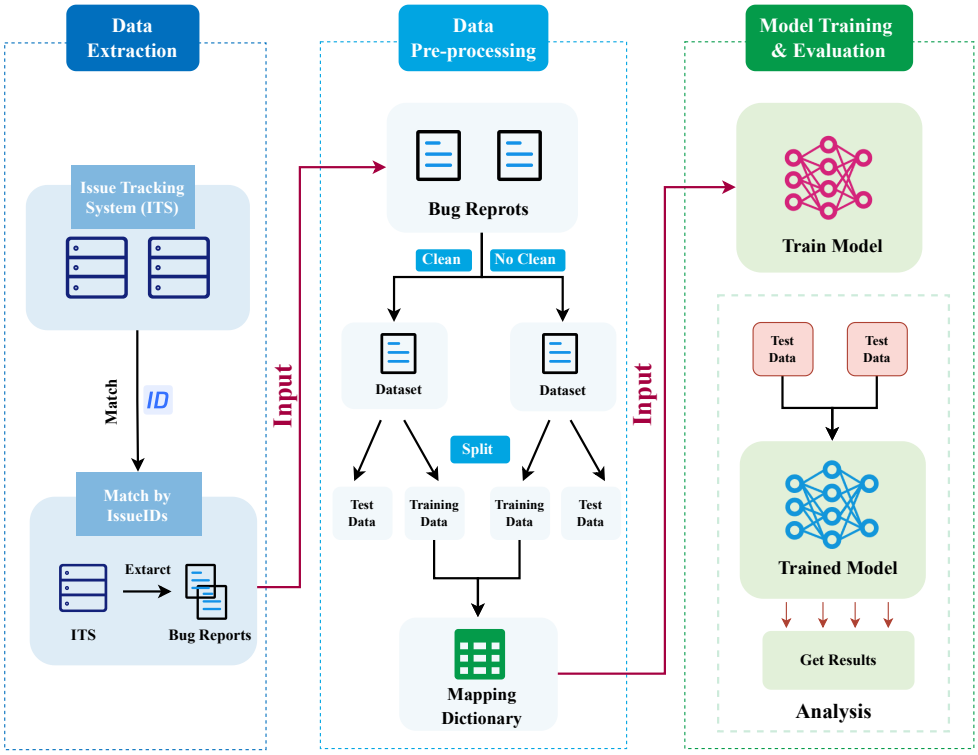


Figure 1: The Overall Framework of Our Experiments

First, we extract bug reports from the VCS based on their identifiers and pre-process data to construct the training dataset. The resulting dataset is then divided into two categories: "code-containing" (for bug reports including code snippets) and "code-absent" (for bug reports without code snippets). Both categories are used in our experiments as follows: For RQ1, we evaluate models using the complete dataset that includes both "code-containing" and "code-absent" bug reports. For RQ2, we specifically use the "code-absent" dataset to train and evaluate models and compare their performance against models trained and evaluated on the complete dataset.

These datasets, along with a mapping dictionary that aligns reports with their respective labels, are used to train the classification models. Finally, the trained models are evaluated on the test set, and we analyze the results.

2.1. Research Questions

Our study aims to explore the effectiveness of using representative deep learning-based methods to predict extrinsic bugs through bug reports. To achieve this, we propose the following research questions:

- **RQ1**: How effective are text classification models in identifying extrinsic bugs through bug reports?
  Motivation: For JIT bug prediction scenarios, misidentifying BICs in extrinsic bugs degrades the quality of training datasets, which in turn affects the performance of JIT models. To ensure high-quality datasets, it is crucial to identify and remove extrinsic bugs. However, manually classifying bug reports is a labor-intensive task. Therefore, we investigate whether text classification methods can be used to automate the identification of extrinsic bugs.

  RQ2: Can datasets that include code as text improve the performance of models in identifying extrinsic bugs?
  Motivation: Prior studies [11, 12, 13] have consistently shown that code snippets and other code-related elements in bug reports, such as stack traces, file/class names, and code examples, provide strong contextual cues for understanding and localizing bugs. Despite this evidence in bug localization research, the role of such code-related textual elements in extrinsic bug identification remains unexplored. In practice, many real-world bug reports consist primarily of code snippets and output logs, which may encode critical signals for distinguishing extrinsic bugs from intrinsic ones. Motivated by these findings, we investigate whether code-containing bug reports can enhance model performance in identifying extrinsic bugs compared to the code-absent dataset.

- **RQ3**: How can the predictions of the proposed CAN model for identifying extrinsic bugs from bug reports be explained?
  Motivation: In RQ1 and RQ2, we evaluated the effectiveness of several models under different scenarios in identifying extrinsic bugs and found that our proposed CAN model achieves competitive performance. However, high predictive accuracy alone is not sufficient for practical adoption, especially in software maintenance scenarios, where developers need to understand why a model makes certain predictions before

trusting and acting on them. Therefore, we focus on the interpretability of the CAN model by employing LIME to analyze which textual features in bug reports contribute most to correct predictions.

2.2. Data Extraction

In the ITS, issue reports can encompass various types, such as bug reports, improvement suggestions, new feature proposals, and other records related to software development. Consistent with existing literature, this study focuses solely on bug reports.

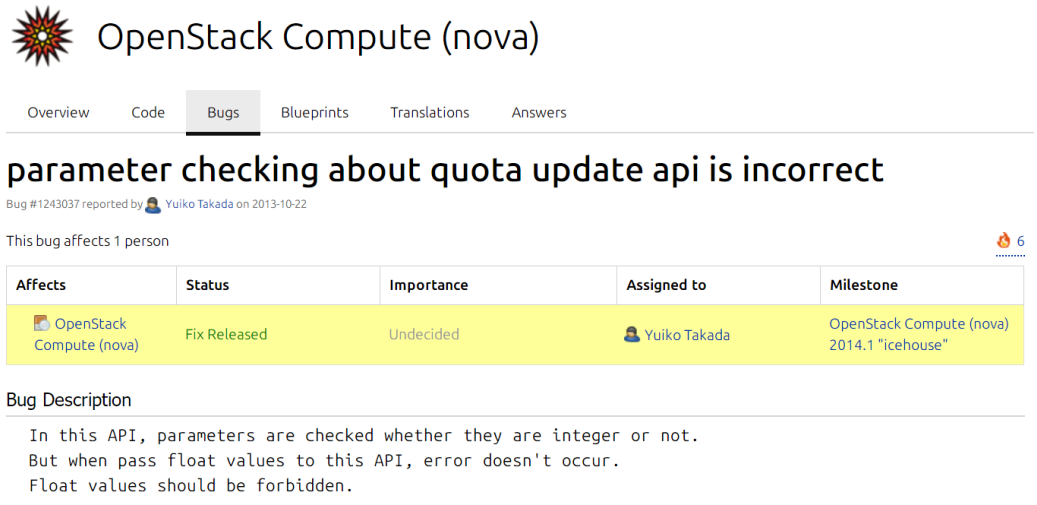
For dataset selection, this study chooses the open-source project OpenStack as the research dataset. This choice is primarily based on previous research findings, particularly the work of Rodríguez-Pérez et al. [3]. They extracted 1,880 bug records from the OpenStack project and invited two reviewers with at least a master's degree in computer science to manually classify these bugs into intrinsic and extrinsic bugs, with the corresponding quantities shown in Table 1. To enhance the consistency and reliability of the classification results, the Krippendorff's Alpha [17] method was employed for reliability assessment during the study, as reported in the original dataset construction work [3].

Component	Total	Intrinsic bug	Extrinsic bug
Swift	31	19	12
Nova	748	490	258
Neutron	648	359	289
Glance	104	43	61
Cinder	349	209	140

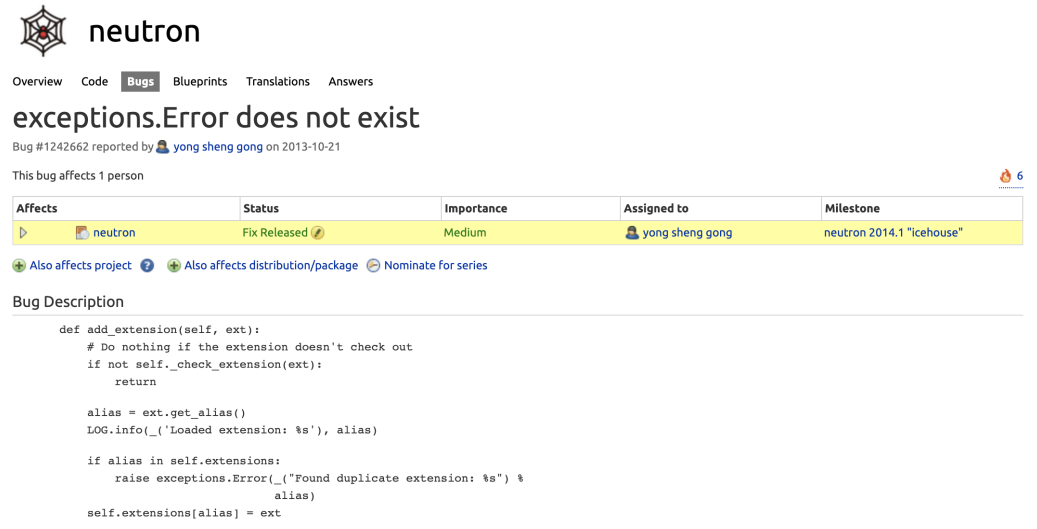
Table 1: The Information of Datasets. Source: [3]

With the information of the 1,880 issues in the ITS and the VCS, we can extract their bug reports for obtaining useful text features. Figure 2(a) illustrates an example of a bug report without code in the OpenStack project, while Figure 2(b) presents an example of a bug report with code in the OpenStack project. In bug reports, the "summary" and "description" fields contain most of the useful information for software analysis. Given the prevalence of bug reports containing code in real-world scenarios, we posit that

such reports can play a crucial role in identifying extrinsic bugs. Accordingly, we extract the textual descriptions from all bug reports to capture relevant features. Subsequently, we construct code-containing and code-absent datasets of bug reports. These datasets will be utilized for further analysis and model training in subsequent phases of our research.



(a) An Example of a Bug Report Without Code in the OpenStack Project. Source: `https://bugs.launchpad.net/nova/+bug/1243037`



(b) An Example of a Bug Report With Code in the OpenStack Project. Source: `https://bugs.launchpad.net/neutron/+bug/1242662`

Figure 2: Examples of Bug Reports in the OpenStack Project

Before training the model, we first performed a statistical analysis of the word count distribution in the bug reports (see Figure 3) to provide a basis for setting the padding length. The results show that most bug reports contain fewer than 150 words, with the highest proportion falling within the 1–49 word range. The shortest report contains 2 words, while the longest contains 1,582 words. The median length is 60 words, and the average length is 107.8 words. Based on this distribution, we selected appropriate padding lengths for subsequent experiments to balance information retention and computational efficiency.

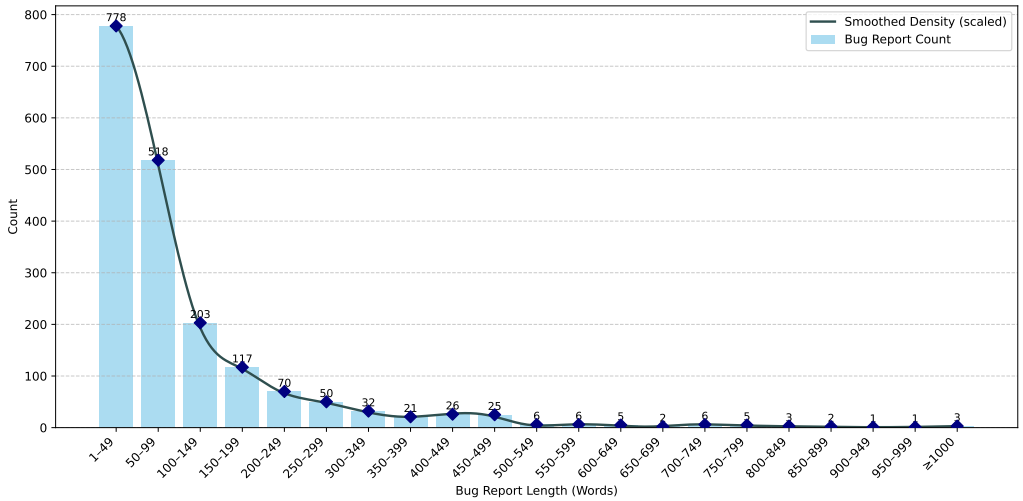


Figure 3: Distribution of Bug Report Lengths (in Words)

To systematically investigate the impact of hyperparameters on model performance, different combinations of batch size and pad size were tested. Specifically, batch size was set to 16 and 32 to compare the effects of small and medium batch sizes on parameter update frequency, memory usage, and gradient estimation stability. At the same time, pad size was set to 64, 128, 256, and 512 based on the bug report length distribution (see Table 2) to evaluate the effect of sequence truncation and padding on model performance.

2.3. Data Pre-processing

This study uses BERT [7], CodeBERT [4], PLBART [5], and CodeT5 [6] as the baseline models. These models, trained on large-scale corpora, exhibit strong language comprehension and are effective at capturing semantic

Max # of Words	Number of Bug Reports	Percentage (%)
64	979	52.07
128	1422	75.64
256	1688	89.78
512	1842	97.98

Table 2: Distribution of Bug Reports by Maximum Number of Words

features and contextual information. As data processing procedures for pre-trained models are generally standardized, this section focuses on the data processing workflow for TextCNN, highlighting its specific application in the text classification task.

In each experiment, the dataset is randomly divided into a training set (60%), a validation set (20%), and a test set (20%). The text data is then standardized and cleaned by removing special characters, punctuation marks, and converting all text to lowercase. Following this preprocessing, words are extracted using spaces as delimiters and sorted by their frequency of occurrence. The 7,000 most frequent words are retained to construct the vocabulary, while low-frequency words are discarded. To evaluate the quality of the constructed vocabulary, pre-trained word embeddings (GloVe) are used for coverage analysis, achieving a coverage rate of 94.36%, thus providing a reliable foundation for subsequent text classification tasks.

During vocabulary extraction and word embedding retrieval, the text is mapped into a machine-readable format using the constructed vocabulary. Words not included in the vocabulary are replaced with special placeholders such as "`<pad>`" (used for padding) and "`<unk>`" (indicating unknown words). Pre-trained word embeddings are then used to generate corresponding vectors for the words in the vocabulary, forming the word embedding matrix. For words that are not present in the pre-trained embeddings, zero vectors or randomly initialized vectors are typically used. To balance performance and storage efficiency, only words in the vocabulary that match pre-trained embeddings are retained, and the processed embeddings are saved as a compact .npz file for efficient use in subsequent model training and evaluation.

2.4. Model Construction

This study compares the performance of the traditional deep learning model TextCNN with several classic pre-trained models (such as BERT, CodeBERT, PLBART, and CodeT5) on the text classification task. Additionally, it draws inspiration from the KAN architecture proposed by Liu et al. [15], which is based on the Kolmogorov-Arnold representation theorem. This theorem states that any multivariate continuous function can be represented as a finite composition of univariate continuous functions, providing a theoretical foundation for constructing neural network architectures with greater interpretability and strong approximation capabilities.

On this basis, we propose a novel model architecture called CAN, which organically integrates TextCNN with KANLinear. The goal is to enhance classification performance and model interpretability in the task of identifying extrinsic bugs. The following sections introduce the basic principles of TextCNN and KANLinear, with a particular focus on the design of the CAN model.

(1) TextCNN

TextCNN [14] is a classical convolutional neural network model for text classification. Its core idea is to use multiple convolution kernels of different sizes to extract features from text and capture local semantic information at various scales. The model first transforms the input text into a word embedding matrix, then applies 2D convolution operations, followed by global max pooling to extract the most salient local features. Finally, these features are passed through a fully connected layer to output the classification result. The model has a simple structure and is computationally efficient, capable of automatically learning key information from text. It is especially suitable for short-text classification tasks and has demonstrated strong performance across various natural language processing scenarios.

(2) KANLinear

The KANLinear model is constructed based on the Kolmogorov-Arnold representation theorem [15]. Its core idea is to transform complex multivariate continuous functions into compositions of multiple univariate continuous functions. The typical structure of the model consists of three stages:

- **Decomposition Stage**: The input multivariate function is first decomposed into several univariate functions. This step leverages the Kolmogorov-Arnold representation theorem.
- **Univariate Function Processing**: Each decomposed univariate function is processed individually. This part can adopt traditional neural network architectures, such as fully connected layers, convolutional layers, etc.
- **Combination Stage**: The processed univariate functions are recombined to produce the final output.

Specifically, for any continuous function $f(x_1, x_2, ..., x_n)$ defined on a closed interval, it can be expressed as a finite combination of univariate continuous additive functions, as shown in Equation 1:

$$f(x_1, x_2, ..., x_n) = \sum_{q=1}^{2n+1} \Phi_q \left(\sum_{p=1}^{n} \psi_{p,q}(x_p) \right) \tag{1}$$

where Φ_q and $\psi_{p,q}$ are continuous univariate functions, and x_i are the input variables.

(3) CAN (Fusion of TextCNN and KANLinear)

To integrate the semantic feature extraction capability of TextCNN with the interpretable structure of KANLinear, this paper proposes a novel model architecture called CAN, as illustrated in Figure 4. The architectural design is as follows:

- **Input Processing Stage**: The input text is first converted into word vectors and fed into the TextCNN module via an embedding layer.
- **Feature Extraction Stage**: Multiple convolution kernels of different sizes are used to perform 2D convolution on the word vectors to extract local features from the text. These features are then compressed through max-pooling to form a fixed-length semantic representation.

- **KANLinear Integration Stage**: The high-dimensional feature vector output from TextCNN is treated as a multivariate input. The structural concept of KANLinear is introduced to perform structured univariate decomposition and recombination of these features. This stage simulates the decomposition and composition mechanisms in the Kolmogorov-Arnold theorem, enhancing the model's ability to represent the intrinsic structure of the features.

- **Output Stage**: The features integrated by the KANLinear module are passed into a fully connected layer and a softmax function to output the final bug category.

This fusion approach not only retains TextCNN's efficient semantic information extraction capability, but also enhances the model's expression and interpretability of high-order feature relationships through KANLinear's structural decomposition.

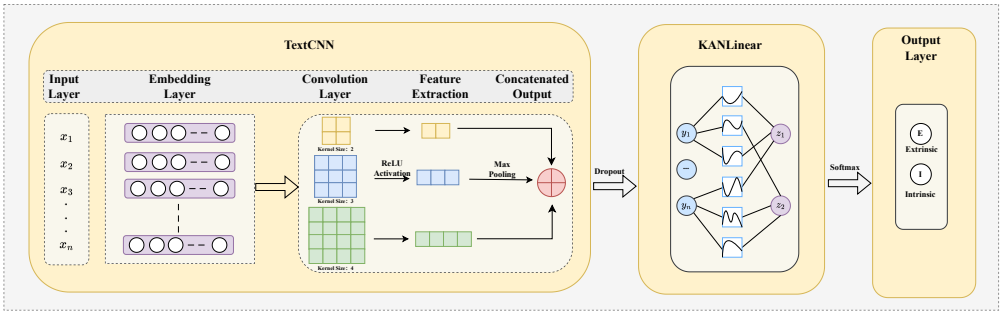


Figure 4: The Framework of CAN Model

2.5. Performance Indicators

To answer RQ1 and RQ2, we evaluate the models using four performance indicators. Their importance varies according to the research focus of this study, identifying extrinsic bug reports, and the degree of class imbalance in the datasets.

2.5.1. F1-Score

In the context of extrinsic bug report identification, the F1-score is the most important metric, as it directly measures the overall ability of the model

to identify extrinsic bug reports (the positive class). It is the harmonic mean of *Precision* and *Recall*:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{2}$$

$$\text{Recall} = \frac{TP}{TP + FN}, \tag{3}$$

$$F1\text{-score} = \frac{2 \times \text{Recall} \times \text{Precision}}{\text{Recall} + \text{Precision}} \tag{4}$$

A higher F1-score indicates a better balance between correctly detecting extrinsic bug reports and avoiding false positives.

2.5.2. Accuracy (ACC)

Accuracy measures the proportion of correctly classified instances among all predictions:

$$ACC = \frac{TP + TN}{TP + TN + FP + FN} \tag{5}$$

2.5.3. Matthews Correlation Coefficient (MCC)

The MCC is a balanced evaluation metric that considers both the positive and negative classes equally, with values in the range $[-1, 1]$. Compared with Accuracy, MCC is more informative when the dataset is highly imbalanced, as it penalizes trivial "predict-all-majority" strategies:

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{6}$$

Since our datasets are not severely imbalanced, as the ratio of positive (extrinsic) to negative (intrinsic) is 2:3, Accuracy can still be a useful overall performance metric. Note that both Accuracy and MCC are less critical than F1.

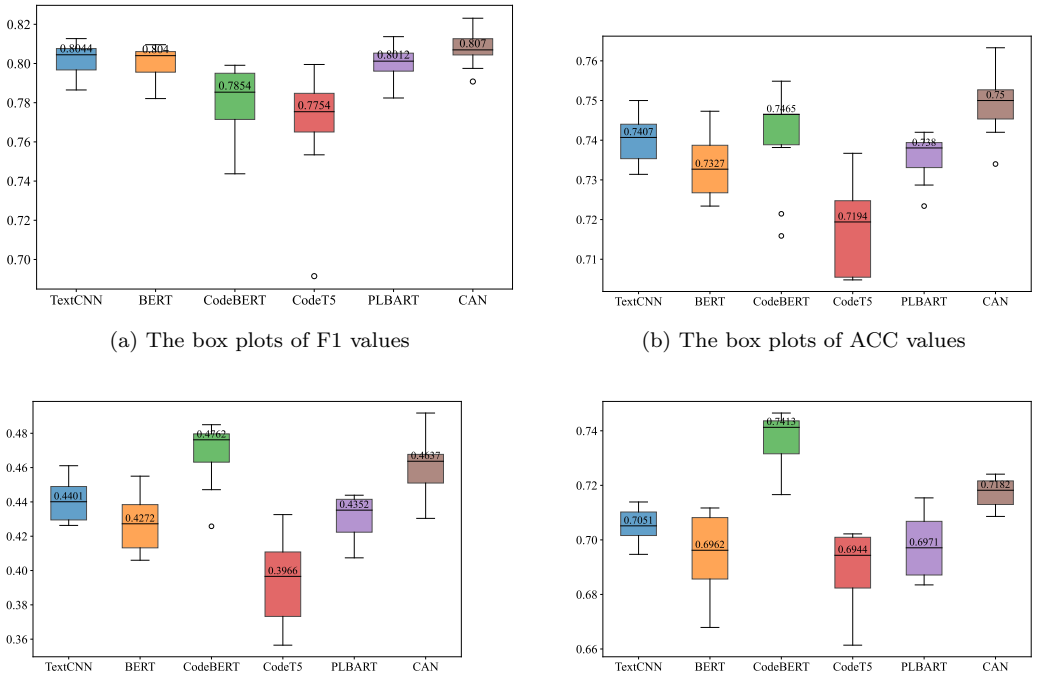
2.5.4. Area Under the ROC Curve (AUC-ROC)

AUC-ROC measures the model's discrimination ability across all classification thresholds. While it is threshold-independent and useful for understanding boundary behavior, studies in bug prediction have noted that AUC-ROC may not directly reflect practical classification performance at a specific decision threshold [18]. In our study, we consider it the least important among the four metrics, though we still report it for completeness.

3. Experimental Results and Analysis

This section addresses the questions raised in Section 3 through experimental results. To investigate the performance of text classification models in identifying extrinsic bugs from bug reports, we employed six different text classification algorithms and applied them to our extracted dataset. To reduce the impact of experimental randomness, we repeated the experiments 10 times. The performance of the models was evaluated using the metrics outlined in Section 4.4.

3.1. RQ1: How effective are text classification models in identifying extrinsic bugs through bug reports?



(a) The box plots of F1 values

(b) The box plots of ACC values

(c) The box plots of MCC values

(d) The box plots of AUC values

Figure 5: Performance of the Six Classification Models

Figure 5 presents box plots of the evaluation metrics for all models under optimal parameters. Note that F1-score is emphasized due to its primary relevance in evaluating extrinsic bug identification performance. From the F1 score box plot, the CAN model achieves the best minimum and maximum

performance with a median of 0.807. TextCNN, BERT and PLBART follow with medians of 0.8044, 0.804 and 0.8012, respectively. CodeBERT and CodeT5 have F1 medians that do not exceed 0.8. From the box plot of ACC, the median accuracy of all models exceeds 0.7. The CAN model performs the best, with a median of 0.75. CodeBERT follows closely with a median of 0.7465, although it has two potential outliers. CodeT5 shows the poorest performance, with a median of only 0.7194. In the AUC box plot, CodeBERT and CAN perform excellently in classifying boundary identification, with median AUCs of 0.7413 and 0.7182, respectively. BERT, CodeT5, and PLBART have relatively lower AUC levels, with medians of 0.6942, 0.6944, and 0.6971, respectively. In the MCC box plot, most models have medians above 0.42, but CodeT5 performs relatively poorly with a median of only 0.3996. CodeBERT and CAN show stronger MCC scores, with medians of 0.4762 and 0.4637, respectively, though CodeBERT has a potential outlier.

In summary, the overall performance of CAN and CodeBERT is the best. The CAN model maintains strong performance across all evaluation metrics, while CodeBERT is particularly outstanding in boundary recognition capability. In contrast, while TextCNN and PLBART perform well in some metrics, their overall stability and classification performance are slightly weaker.

Table 3: Statistical Significance and Effect Size between CAN and CodeBERT

Metric	p-value	Significance	Effect Size r	Effect Level
F1-score†	0.0003	***	0.7691	Large
Accuracy†	0.0513	ns	0.3719	Medium
MCC‡	0.1061	ns	0.2874	Small
AUC-ROC‡	0.0004	***	0.7606	Large

† CAN vs. CodeBERT, ‡ CodeBERT vs. CAN

Significance markers: most significant (***) $p < 0.001$, highly significant (**) $0.001 \leq p < 0.01$', significant (*) $0.01 \leq p < 0.05$, otherwise "ns" (no significance).

Effect size levels: Large $r > 0.5$, Medium $0.3 < r \leq 0.5$, Small $0.1 < r \leq 0.3$, otherwise Negligible.

To statistically validate the performance difference between CAN and CodeBERT, we performed a **one-sided non-parametric Wilcoxon Rank-Sum test** (Mann–Whitney U Test) across 10 repeated runs and computed the **effect size** r. Specifically, r is computed from the standardized z-score of the Wilcoxon Rank-Sum test using the formula $r = \frac{|z|}{\sqrt{N}}$, where N denotes

the total number of observations across both groups.

The results (Table 3) show that CAN's F1-score is significantly higher than CodeBERT's ($p = 0.0003$, ***, $r = 0.7691$, Large effect). This provides strong statistical evidence that CAN is better at identifying extrinsic bug reports. In contrast, Accuracy and MCC differences between CAN and CodeBERT are not statistically significant, suggesting a comparable overall classification balance. For AUC-ROC, CodeBERT significantly outperforms CAN ($p = 0.0004$, Large effect). However, as discussed in Section 2.5, this advantage mainly reflects potential performance at alternative thresholds and does not outweigh CAN's superiority in F1-score at the standard 0.5 threshold.

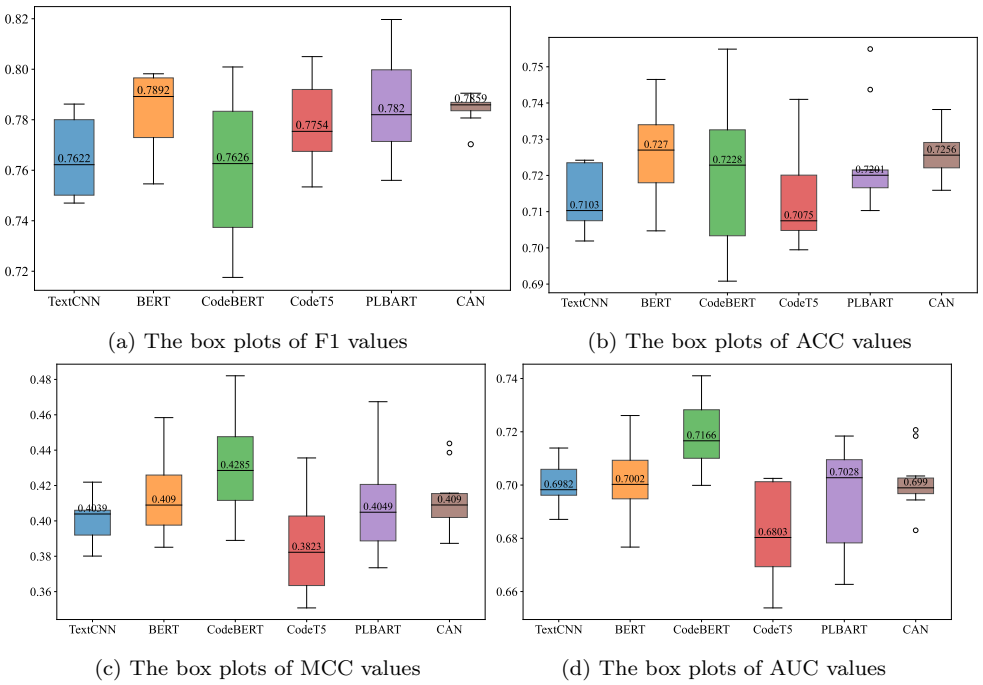
Overall, while CodeBERT benefits from pretraining on large-scale code–text corpora (potentially including overlaps with OpenStack data), our from-scratch CAN model achieves significantly better F1-score in this task, indicating stronger practical utility for extrinsic bug report identification.

RQ1 Conclusion: Text classification models can effectively identify extrinsic bugs from bug reports. CAN demonstrates the best performance in terms of the most important metric (F1-score), with a statistically significant and large effect size advantage over CodeBERT, despite the latter's edge in AUC-ROC.

3.2. RQ2: Can datasets that include code as text improve the performance of models in identifying extrinsic bugs?

To analyze RQ2, we conducted experiments using both code-containing and code-absent datasets. We performed 10 rounds of experiments for each, using consistent model parameters as described in Section 3.1. The median values of each metric were computed and compared.

We observe from the box plot (Figure 6) that the CAN model demonstrates strong stability, despite the presence of multiple outliers. In contrast, the CodeBERT model exhibits greater variability in performance, particularly in the ACC and F1 metrics, where fluctuations are significant. For the F1 score, no model achieves a median above 0.8. The BERT model performs the best, with a median F1 score of 0.7892, followed by CAN at 0.7859. TextCNN and CodeBERT perform worse, with median F1 scores of 0.7622 and 0.7626, respectively. In terms of ACC, all models achieve a median accuracy above 0.7. Among these models, BERT performs the best, with



(a) The box plots of F1 values

(b) The box plots of ACC values

(c) The box plots of MCC values

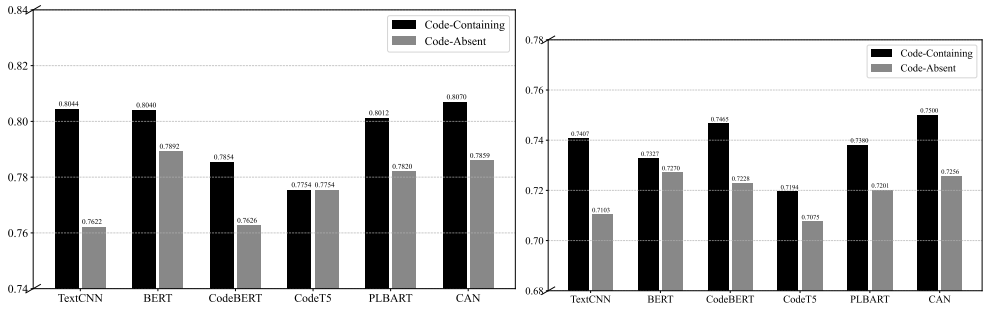
(d) The box plots of AUC values

Figure 6: Performance of the Six Classification Models on Code-Absent Dataset

a median ACC of 0.7270, followed by CAN. The CodeT5 model performs the worst, with a median ACC of only 0.7075. Regarding AUC, the CodeBERT model achieves the highest median AUC of 0.7166, followed closely by PLBART with 0.7028 and BERT with 0.7002. Other models remain below 0.7 in this metric. Notably, the CAN model has three outliers, potentially impacting its overall stability. In terms of MCC, most models achieve a median above 0.40. The CodeT5 model performs the worst, with a median MCC of only 0.3823. CodeBERT outperforms other models in this metric, achieving a median MCC of 0.4285. Both BERT and CAN have a median MCC of 0.409, demonstrating relatively strong overall performance.

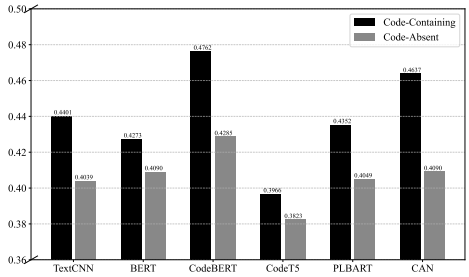
To further analyze the impact of source code on model performance, we use histograms for comparison, as shown in Figure 7.

The results from the histogram and data analysis clearly indicate that the performance of all models has deteriorated in F1 and ACC scores, with TextCNN, CAN, and CodeBERT exhibiting the most substantial declines. This suggests that code data was instrumental in the training and classification of these models. In terms of AUC, BERT and PLBART exhibit slight

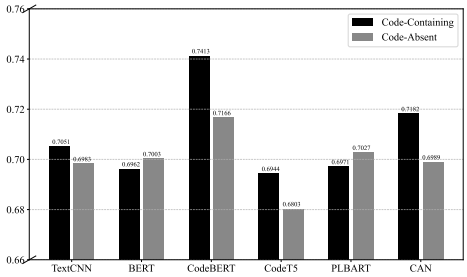


(a) The bar chart of F1 values

(b) The bar chart of ACC values



(c) The bar chart of MCC values



(d) The bar chart of AUC values

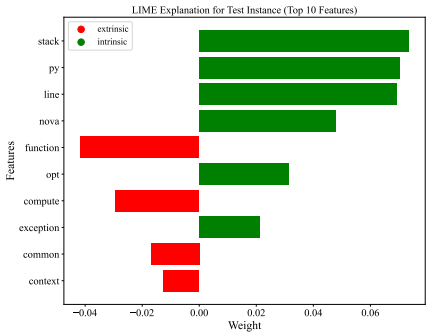
Figure 7: Model Performance Comparison on Code-Containing vs. Code-Absent Datasets

improvements; however, the overall impact is marginal and does not offset the declines observed in other metrics.

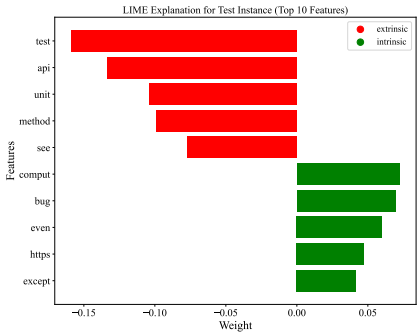
RQ2 Conclusion: The experimental results indicate that incorporating code into the dataset enhances all models' ability to identify extrinsic bugs. Conversely, removing source code data detrimentally impacts the overall performance of the models.

3.3. RQ3: How can the predictions of the proposed CAN model for identifying extrinsic bugs from bug reports be explained?

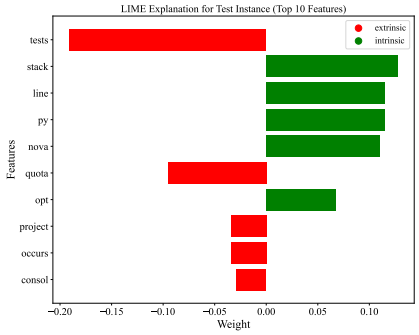
To address research question RQ3, we employed the LIME method to evaluate the feature importance of correctly predicted instances. Bug reports containing source code were selected as the dataset, which was then divided into training and test sets. The CAN model was subsequently trained using this dataset. After training, each instance in the test dataset was fed into the model, and LIME was utilized to extract and interpret the ten most important textual features.



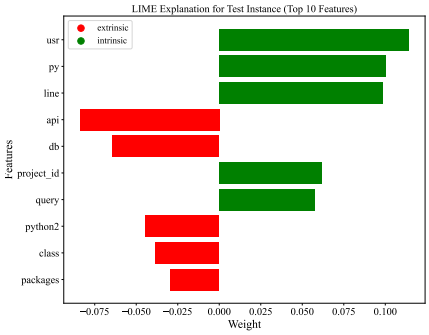
(a)



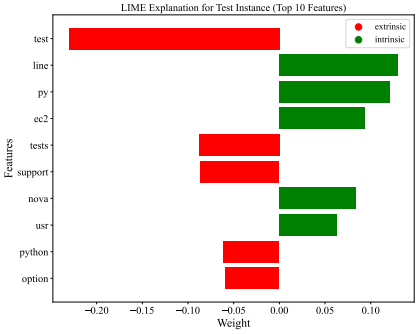
(b)



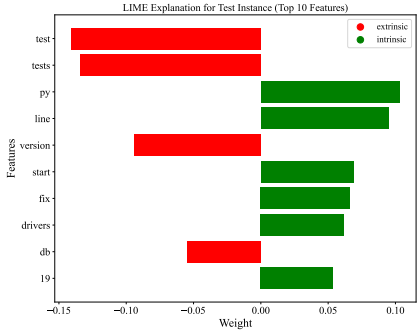
(c)



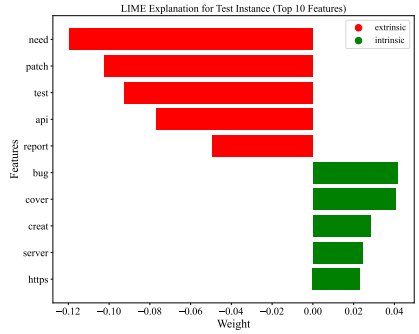
(d)



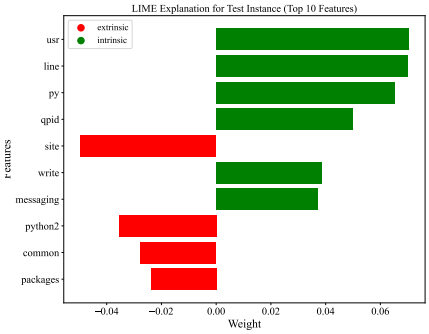
(e)



(f)



(g)



(h)

Figure 8: The Text Features Distribution Diagram of 8 Instances Predicted by CAN Model

By analyzing the visualization results of feature explanations for all instances in the test dataset (as shown in Figure 8), it was observed that certain specific terms had a significant impact on predicting extrinsic bugs, such as "test" illustrated in subfigures (b), (f), and (g); "tests" in subfigures (c), (e), and (f), "api" in subfigures (b) and (g). In contrast, words like "py" and "line" were more likely to lead the model toward predicting intrinsic bugs, as shown in subfigures (a), (d), and (h). Further examination of several bug reports revealed that these key textual features frequently appeared in extrinsic bug reports, thereby playing a crucial role in the model's prediction outcomes.

We subsequently conducted a detailed examination of these specific bug report instances. The results indicate that: "test" often appears in contexts related to testing, suggesting that the bug may be due to changes in test cases or test environments. "api" refers to interactions between extrinsic systems or services and the software. Bug reports frequently describe problems such as API call failures or interface incompatibilities. "py" is commonly found in contexts related to Python code or scripts. Intrinsic bugs often involve issues in code implementation or errors discovered during the development process. "line" often refers to specific code line numbers mentioned in bug reports, which helps developers quickly locate the issue in the code.

RQ3 Conclusion: Experimental results indicate that words such as "line" and "py" significantly impact the prediction of intrinsic bugs, while terms like "test" and "api" have a substantial influence on the prediction of extrinsic bugs.

4. Discussion and Implication

4.1. Discussion1: The impact of batch size and pad size on model performance

This experiment investigates the impact of batch size and pad size on model performance. Different combinations of batch size and pad size were tested, as detailed below:

- **batch size**: 16, 32
- **pad size**: 64, 128, 256, 512

Multiple rounds of experiments were conducted to observe their effects on the models' F1 score (the primary evaluation metric), as well as on ACC, MCC, and AUC. The specific experimental results are shown in Table 4. The **bold** values indicate the best performance achieved under each metric within the tested settings. Due to GPU resource limitations, the experiment with a batch size of 32 and a pad size of 512 could not be completed, and data for this combination is missing.

From the analysis of the experimental data, the following conclusions can be drawn:

- **Pad size has a significant impact on model performance**: In most models, as pad size increases, F1, ACC, MCC, and AUC metrics tend to rise. Specifically, CodeBERT, CodeT5, and CAN models perform best with a pad size of 256, but when pad size is increased to 512, performance decreases, likely due to the introduction of more padding, which increases noise. However, TextCNN, PLBART, and BERT models continued to improve as pad size increased further.

- **The impact of batch size is relatively small**: Overall, models performed better with a batch size of 16 compared to a batch size of 32. This may be because a smaller batch size helps the model learn sample features in more detail, but it also increases training time.

4.2. Discussion2: The effectiveness of each component in our proposed CAN model

To comprehensively evaluate the actual contribution of each key component in the CAN model to overall performance, we designed a series of ablation experiments. Under a consistent input configuration, we either replaced or removed the TextCNN feature extraction module and the KANLinear classification module, and observed the performance variations, thereby verifying the impact of each sub-module on the final classification results.

Table 5 presents the performance comparison of different combinations of feature extraction and classification modules. Similarly, **bold** values mark the highest scores obtained for each metric. Overall, the combination of TextCNN + KANLinear achieved the best results across all evaluation metrics (F1, ACC, MCC, AUC), with an F1 score of 0.8072, an accuracy of 0.7492, an MCC of 0.4610, and an AUC of 0.7175. When replacing the KANLinear classifier with a simple linear layer, the TextCNN-based model still

Table 4: Impact of Batch Size and Pad Size on Model Performance

Model	batch size	pad size	ACC	AUC	F1	MCC
BERT	16	64	0.7117	0.6839	0.7730	0.3864
		128	0.7120	0.6889	0.7697	0.3911
		256	0.7239	0.6884	0.7891	0.4059
		512	**0.7335**	0.6943	**0.7996**	**0.4266**
	64	64	0.6819	0.6563	0.7331	0.3410
		128	0.7011	0.6786	0.7604	0.3681
		256	0.7296	**0.6973**	0.7916	0.4167
CodeBERT	16	64	0.6905	0.7003	0.7222	0.3915
		128	0.7259	0.7332	0.7590	0.4546
		256	**0.7410**	**0.7375**	0.7810	**0.4682**
		512	0.7335	0.6943	**0.7996**	0.4266
	64	64	0.6975	0.6931	0.7426	0.3817
		128	0.7228	0.7145	0.7692	0.4244
		256	0.7153	0.6931	0.7724	0.3961
CodeT5	16	64	0.7122	0.6753	**0.7811**	0.3789
		128	0.7128	**0.6906**	0.7705	0.3886
		256	**0.7181**	0.6899	0.7696	**0.3950**
		512	0.7138	0.6891	0.7727	0.3913
	64	64	0.6612	0.6078	0.7536	0.2512
		128	0.6976	0.6641	0.7672	0.3469
		256	0.6856	0.6474	0.7612	0.3166
PLBART	16	64	0.7107	0.6893	0.7794	0.3868
		128	0.7150	0.6902	0.7836	0.3979
		256	0.7210	0.6894	0.7850	0.3998
		512	**0.7357**	**0.6979**	**0.8001**	**0.4309**
	64	64	0.7023	0.6865	0.7658	0.3709
		128	0.7112	0.6859	0.7761	0.3845
		256	0.7211	0.6950	0.7844	0.4015
TextCNN	16	64	0.7284	0.6929	0.7932	0.4143
		128	0.7260	0.6973	0.7865	0.4120
		256	0.7391	0.7056	0.8004	0.4381
		512	**0.7404**	0.7051	**0.8026**	**0.4410**
	64	64	0.7269	0.6933	0.7904	0.4123
		128	0.7327	0.6955	0.7977	0.4236
		256	0.7369	**0.7088**	0.7943	0.4394
CAN	16	64	0.7207	0.6860	0.7866	0.3966
		128	0.7269	0.6987	0.7868	0.4140
		256	**0.7492**	**0.7175**	**0.8072**	**0.4610**
		512	0.7468	0.7147	0.8057	0.4554
	64	64	0.7208	0.6978	0.7775	0.4064
		128	0.7282	0.6966	0.7903	0.4148
		256	0.7399	0.7121	0.7972	0.4420

Table 5: Effectiveness of Each Component in the CAN Model

Feature Extractor	Classifier	ACC	AUC	F1	MCC
TextCNN	**KANLinear**	**0.7492**	**0.7175**	**0.8072**	**0.4610**
TextCNN	Linear	0.7404	0.7051	0.8026	0.4410
DPCNN	KANLinear	0.7073	0.6691	0.7797	0.3547
DPCNN	Linear	0.6813	0.6518	0.7646	0.3222

maintained relatively high performance (e.g., F1 = 0.8026, ACC = 0.7404), but all metrics showed slight declines, indicating that the KANLinear module indeed enhances the model's classification capability.

Furthermore, we also introduced DPCNN [19] as an alternative feature extractor to validate the generality of the KANLinear module. DPCNN (Deep Pyramid Convolutional Neural Network) is a low-complexity word-level deep CNN architecture designed for text categorization, capable of capturing long-range dependencies and global text representations through a pyramidal structure with relatively low computational cost. It has been shown to outperform several state-of-the-art models on benchmark datasets for sentiment classification and topic categorization. In our experiments, DPCNN + KANLinear outperformed DPCNN + Linear, particularly in ACC and MCC (improving by 3.82% and 10.09%, respectively), further confirming the effectiveness of the KANLinear classifier across different feature extractors. However, the overall performance of DPCNN in our task was still lower than TextCNN, which may be due to weaker structural suitability for the specific characteristics of bug reports classification.

In summary, the results demonstrate that TextCNN and KANLinear play crucial roles in feature extraction and classification modeling, respectively. Their combination yields the best classification performance, thereby validating the rationality and effectiveness of the CAN model architecture.

4.3. Implication

Our findings indicate that text classification models are highly effective in identifying and predicting extrinsic bugs. Among them, the CAN model performed the best, achieving an ACC of 0.7492 and an F1 score of 0.8072. Consequently, this model was utilized in RQ3 experiments. Analysis of feature importance for test dataset instances revealed that certain key textual features significantly affect the prediction of extrinsic/intrinsic bugs. As illustrated in Figure 8, these crucial semantic features can differentiate between

intrinsic and extrinsic bugs but may sometimes lead to incorrect predictions. This necessitates further exploration into text feature engineering and model tuning to enhance robustness and accuracy.

Moreover, our research confirms that including source code in bug reports can enhance model prediction performance. Bug reports created in ITSs often include descriptions and stack traces (optional). The description, recorded by the submitter, explains observed behavior during a crash and details the bug. Stack traces are a series of function calls from the memory stack during the crash. In practice, many bug report submitters only include stack traces or source code as the bug description. Nayrolles et al. [20] demonstrated that stack traces are useful for understanding bugs. Inspired by this, and given that stack traces may provide valuable information, we hypothesized that other technical details (e.g., source code) might also aid in identifying extrinsic bugs and designed experiments accordingly.

It is important to note that our approach might not effectively extract useful information from source code or stack traces that aids in identifying extrinsic bugs. We treated the source code merely as text, meaning that its deeper semantic information has not been fully explored. Programs have specific syntactic structures and rich semantic information hidden in abstract syntax trees (AST), which are crucial representations of source code. Therefore, we believe further exploration is needed to utilize structural information within source code. We need to find corresponding links in the software system to more accurately determine whether a bug is intrinsic or extrinsic, rather than treating the source code as plain text.

Over the past 20 years, many studies have used automated techniques to collect JIT bug prediction datasets to address bug repair submissions, bug-introducing changes, and bug reports. These datasets have been used to train JIT bug prediction models. However, Rodríguez Pérez et al. [3] demonstrated that JIT bug prediction models trained only on intrinsic bugs can more accurately reflect real-world scenarios, as indicated by different (usually higher) AUC values. This suggests that extrinsic bugs should be excluded to reduce the predictive capacity loss of JIT bug prediction models. Consequently, hundreds of studies on JIT bug prediction [21] might be less accurate than expected, as they did not use data distinguishing between intrinsic and extrinsic bugs for model training. Researchers and practitioners should ensure high-quality data to avoid noise affecting model training. Therefore, we recommend adding an intrinsic/extrinsic field to new bug reports and using better-performing classifiers to automatically generate and

label historical data.

5. Threats to Validity

5.1. Internal Validity

The first threat to internal validity primarily stems from potential errors in our experiments and biases introduced by randomization. To address this threat, we manually checked the completeness of the extracted bug reports and the labels of the datasets used in the study. We then re-evaluated the implementation process and reported the average performance across 10 experiments. Additionally, issues related to the mapping dictionary we designed also pose a threat. Upon examining the bug reports in the dataset, we found many stop words, such as "hello", "why", and "how", that do not provide useful textual features and negatively affect the prediction of extrinsic bugs. These stop words were removed before mapping string tokens.

5.2. External Validity

The primary threat to external validity concerns the generalizability of our research findings. Our dataset consists of only 1,880 issues from the OpenStack project and was manually labeled by two master's students specializing in Computer Science and Technology. Due to the relatively small size of the dataset and its extraction from a single project, the consistency of the labels requires further investigation. Moreover, since OpenStack is a large-scale Python-based open-source project, the results may be biased toward the characteristics of Python projects. This limits the generalizability of our findings to projects developed in other programming languages such as Java, C++, or JavaScript, where bug report structures, vocabulary, and code-related expressions may differ. Additionally, involving more experts in the labeling process and applying semi-supervised or active learning strategies will help improve the reliability and scalability of the labeling process in diverse contexts.

5.3. Construct Validity

Threats to construct validity concern whether the evaluation metrics accurately reflect the prediction performance of classifiers. ACC reflects the proportion of correct predictions made by the model. However, this metric assumes that different classes are balanced, meaning it becomes unreliable when the dataset is imbalanced. F1-score is the harmonic mean of precision

and recall, but it cannot fully address cases where one model has high precision but low recall, while another has low precision but high recall, yet both achieve similar F1-scores. In contrast, AUC and MCC can mitigate these issues, as they are not sensitive to class imbalances and do not depend on threshold selection. Therefore, we use four indicators—F1, ACC, MCC, and AUC—to comprehensively evaluate the performance of the models.

Another construct validity threat relates to the interpretability method used in our study. We adopted LIME primarily because of its adoption in prior JIT bug prediction research (e.g., JITLine) and supporting empirical findings that classifier-agnostic methods such as LIME, SHAP [22], and BreakDown [23] can yield consistent feature importance rankings [24]. While we acknowledge that alternative methods like SHAP offer stronger theoretical grounding and may produce more stable explanations, incorporating multiple explainers would require significant additional experimental effort and computational resources. Given that our LIME-based results, combined with manual inspection, produce reasonable and coherent explanations, we did not re-implement the analysis with other interpretability tools. We explicitly acknowledge this as a limitation to encourage future studies to perform more comprehensive and systematic comparative evaluations of multiple explanation techniques in this domain.

6. Related Work

6.1. Just-In-Time Bug Prediction

The concept of Just-In-Time (JIT) bug prediction was first introduced by Mockus and Weiss [25], who designed change metrics to predict whether a software change would introduce a bug. Traditional approaches relied heavily on machine learning models that assumed the characteristics of past and future bug-introducing changes (BICs) are similar. Early studies extracted predictive attributes from version control systems (VCSs), issue tracking systems (ITSs) [26, 27, 28], and code review systems [29, 30] to train classifiers [31, 32].

With the rise of deep learning, models such as DeepJIT, the first deep learning framework for JIT prediction based on TextCNN [14], and JITGNN, a graph neural network-based approach, demonstrated the ability to capture complex structural and semantic information in code changes. More recent methods have leveraged pre-trained models of code to capture richer semantic

and contextual information. JIT-Fine [33] and JIT-Smart [34] adopt CodeBERT [4] multi-task learning to jointly address bug prediction and localization, significantly improving performance over traditional baselines. Later, general-purpose pre-trained models of code such as PLBART [5] and CodeT5 [6] have shown strong performance across code understanding and generation tasks. These models represent distinct categories of deep learning approaches and provide strong baselines for evaluating text classification in software engineering, making them natural choices for our experiments designed for **RQ1** and **RQ2**.

Interpretability has also gained attention in JIT bug prediction. JITLine, for example, applied LIME to generate line-level bug localization explanations in addition to predictions. Yang et al. [24] conducted a large-scale empirical study showing that classifier-agnostic techniques such as LIME, BreakDown, and SHAP yield consistent feature importance rankings, reinforcing their reliability. This interpretability perspective provides the methodological basis for our **RQ3**, where we use LIME to explain the predictions of our proposed CAN model.

6.2. Bug Reports and Extrinsic Bug Identification

Beyond code change metrics, bug reports in ITS contain rich textual information that can be leveraged for bug prediction. A growing body of work has examined the classification of bug origins, the quality of bug reports, and their impact on predictive modeling.

Several studies [1, 2, 3] distinguish between *intrinsic bugs* (originating from internal code changes and associated with a BIC) and *extrinsic bugs* (originating from external factors such as API changes, dependency updates, or environmental variations, lacking a BIC but having a first-failing change). These works highlight the limitations of SZZ-based BIC identification and demonstrate that removing extrinsic bugs from datasets can significantly improve JIT model performance. Our **RQ1** is directly motivated by these findings: rather than manually filtering extrinsic bugs, we aim to automatically identify them from bug report contents.

Another related challenge is misclassification in bug reports for bug prediction and localization. Prior studies [35, 36] show that reports mislabeled as bugs but actually describing feature requests, documentation updates, or other maintenance tasks can distort both bug prediction and localization results. Understanding and mitigating such noise is essential for building robust classification models.

The quality of bug reports has also been studied extensively. High-quality reports, those including steps to reproduce stack traces, and particularly code examples, have been shown to facilitate bug localization and reduce fixing time [11, 12, 13]. For example, Zimmermann et al. [11] found that developers favored bug reports containing code snippets or stack traces, as these structured elements often clarified the underlying cause of a bug. Similarly, Mills et al. [12] demonstrated that bug reports with explicit localization hints, including code fragments, significantly improved the accuracy of bug localization tasks. Furthermore, Zhang et al. [13] reported that most contributors for both desktop software and mobile apps believed that stack traces, code examples, and patches shortened bug-fixing time and revealed bug causes more clearly. These insights form the foundation for our **RQ2**, where we investigate whether incorporating code as text in datasets improves extrinsic bug identification.

In conclusion, previous works have either (1) classified bug origins and analyzed their impact on prediction models [1, 2, 3], (2) studied quality indicators in bug reports [11] and their role in localization performance [12, 13], or (3) investigated the negative effects of mislabeled reports [35, 36]. However, none has addressed the automatic identification of extrinsic bugs from bug reports using modern text classification models. Our work fills this gap by systematically exploring the effectiveness of representative deep learning-based classification models in extrinsic bug classification, examining the effect of including code snippets in bug report contents, and applying LIME to explain predictions of our proposed CAN model combining TextCNN and KAN.

7. Conclusion and Future Work

Bug reports are created by software developers within an ITS to describe technical details such as the cause, symptoms, severity, and other features of bugs. An ITS includes bug reports for both intrinsic and extrinsic bugs. Intrinsic bugs refer to errors with explicit BICs in the VCS. Extrinsic bugs are primarily caused by external factors of the project and cannot be matched with a BIC.

Rodríguez-Pérez et al. [3] demonstrated through a case study of the OpenStack system that intrinsic and extrinsic bugs are different. JIT bug prediction models trained only on intrinsic bugs can more accurately reflect the real world, as indicated by higher AUC values. This suggests that ex-

trinsic bugs should be excluded to minimize the loss of predictive power in JIT bug prediction models. Consequently, hundreds of studies on JIT bug prediction [21] may not be as accurate as possible because they did not use data distinguishing between intrinsic and extrinsic bugs to train their models. Manual analysis of bugs and their reports is labor-intensive and time-consuming, and models trained solely on intrinsic bugs typically achieve higher AUC scores. Therefore, we use various text classification techniques to automate the classification of bugs into intrinsic or external categories. Our experiments validate the effectiveness of different text classification models in distinguishing between intrinsic and extrinsic bugs. The results show that incorporating source code in bug reports leads to improved classification performance. Specifically, the CAN model performed the best in classification, with an ACC of 0.7492 and an F1 score of 0.8072.

Our research also indicates that certain key semantic features play a significant role in distinguishing between intrinsic and extrinsic bugs. For example, words like "test" and "api" have a notable impact on predicting extrinsic bugs, while "line" and "py" are more influential in predicting intrinsic bugs. Additionally, the results reveal that source code information can help identify extrinsic bugs in some cases, but its deeper semantic information has not been fully exploited. Future research should further explore how to leverage structural information in source code.

Future work includes: (1) Applying our method to datasets from other ecosystems, including Java-based projects (e.g., JDT, Platform) and cross-language repositories, to evaluate the robustness of the proposed model across different language domains; (2) Utilizing more semi-supervised and unsupervised learning methods to identify extrinsic bugs, thereby reducing the cost of manual labeling; (3) Exploring the effectiveness of our methods in real-world development scenarios.

Acknowledgments

This work was partially supported by the National Natural Science Foundation of China (No. 62372174), the Computational Biology Program of Shanghai Science and Technology Commission (No. 23JS1400600), the Capacity Building Project of Local Universities Science and Technology Commission of Shanghai Municipality (No. 22010504100), the Research Programme of National Engineering Laboratory for Big Data Distribution and Exchange Technologies (No. 2021-GYHLW-01007), and the Shanghai 2024

Science and Technology Innovation Action Plan Star Cultivation (Sailing Program, No. 24YF2719900 and 24YF2720000).

References

[1] G. Rodríguez-Pérez, A. Zaidman, A. Serebrenik, G. Robles, J. M. González-Barahona, What if a bug has a different origin?: Making sense of bugs without an explicit bug introducing change, in: Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, 2018, p. 52.

[2] G. Rodríguez-Pérez, G. Robles, A. Serebrenik, A. Zaidman, D. M. German, J. M. Gonzalez-Barahona, How bugs are born: A model to identify how bugs are introduced in software components, Empirical Software Engineering 25 (2020) 1294–1340.

[3] G. Rodríguez-Pérez, M. Nagappan, G. Robles, Watch out for extrinsic bugs! A case study of their impact in Just-In-Time bug prediction models on the openstack project, IEEE Transactions on Software Engineering 48 (2022) 1400–1416.

[4] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, M. Zhou, CodeBERT: A pre-trained model for programming and natural languages, in: Findings of the Association for Computational Linguistics: EMNLP 2020, 2020, pp. 1536–1547.

[5] W. Ahmad, S. Chakraborty, B. Ray, K.-W. Chang, Unified pre-training for program understanding and generation, in: Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, 2021, pp. 2655–2668.

[6] Y. Wang, W. Wang, S. Joty, S. C. Hoi, CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, in: Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, 2021, pp. 8696–8708.

[7] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, BERT: Pre-training of deep bidirectional transformers for language understanding, in: Proceedings of the 2019 Conference of the North American Chapter of the

Association for Computational Linguistics: Human Language Technologies, 2019, pp. 4171–4186.

[8] T. Huang, H.-Q. Yu, G.-S. Fan, Z.-J. Huang, C.-Y. Wu, A code change-oriented approach to just-in-time defect prediction with multiple input semantic fusion, Expert Systems 41 (2024) e13702.

[9] J. Svajlenko, C. K. Roy, Bigclonebench: A retrospective and roadmap, in: 2022 IEEE 16th International Workshop on Software Clones (IWSC), 2022, pp. 8–9.

[10] Y.-G. Liang, G.-S. Fan, H.-Q. Yu, M.-C. Li, Z.-J. Huang, Automatic code summarization using abbreviation expansion and subword segmentation, Expert Systems 42 (2025) e13835.

[11] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schroter, C. Weiss, What makes a good bug report?, IEEE Transactions on Software Engineering 36 (2010) 618–643.

[12] C. Mills, E. Parra, J. Pantiuchina, G. Bavota, S. Haiduc, On the relationship between bug reports and queries for text retrieval-based bug localization, Empirical Software Engineering 25 (2020).

[13] T. Zhang, J. Chen, X. Luo, T. Li, Bug reports for desktop software and mobile apps in github: What's the difference?, IEEE Software 36 (2019) 63–71.

[14] Y. Kim, Convolutional neural networks for sentence classification, in: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, 2014, pp. 1746–1751.

[15] Z. Liu, Y. Wang, S. Vaidya, F. Ruehle, J. Halverson, M. Soljacic, T. Y. Hou, M. Tegmark, Kan: Kolmogorov-arnold networks, in: The Thirteenth International Conference on Learning Representations, 2025.

[16] M. T. Ribeiro, S. Singh, C. Guestrin, "why should i trust you?": Explaining the predictions of any classifier, in: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Association for Computing Machinery, New York, NY, USA, 2016, p. 1135–1144.

[17] K. Krippendorff, Estimating the reliability, systematic error and random error of interval data, Educational and Psychological Measurement 30 (1970) 61–70.

[18] R. Moussa, F. Sarro, On the use of evaluation measures for defect prediction studies, in: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, 2022, p. 101–113.

[19] R. Johnson, T. Zhang, Deep pyramid convolutional neural networks for text categorization, in: Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, Vancouver, Canada, 2017, pp. 562–570.

[20] M. Nayrolles, A. Hamou-Lhadj, S. Tahar, A. Larsson, JCHARMING: A bug reproduction approach using crash traces and directed model checking, in: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, 2015, pp. 101–110.

[21] L. Cai, Y. Fan, M. Yan, X. Xia, Just-In-Time software defect prediction: Literature review, Journal of Software 30 (2019) 1288–1307.

[22] M. Scott, L. Su-In, et al., A unified approach to interpreting model predictions, Advances in neural information processing systems 30 (2017) 4765–4774.

[23] M. Staniak, P. Biecek, Explanations of model predictions with live and breakdown packages, The R Journal 10 (2018) 395–409.

[24] X. Yang, H. Yu, G. Fan, Z. Huang, K. Yang, Z. Zhou, An empirical study of model-agnostic interpretation technique for Just-In-Time software defect prediction, in: Collaborative Computing: Networking, Applications and Worksharing: 17th EAI International Conference, CollaborateCom 2021, Virtual Event, October 16-18, 2021, Proceedings, Part I 17, 2021, pp. 420–438.

[25] A. Mockus, D. M. Weiss, Predicting risk of software changes, Bell Labs Technical Journal 5 (2000) 169–180.

[26] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, N. Ubayashi, A large-scale empirical study of Just-In-Time quality assurance, IEEE Transactions on Software Engineering 39 (2013) 757–773.

[27] T. L. Graves, A. F. Karr, J. S. Marron, H. Siy, Predicting fault incidence using software change history, IEEE Transactions on Software Engineering 26 (2000) 653–661.

[28] S. Kim, E. J. Whitehead, Y. Zhang, Classifying software changes: Clean or buggy?, IEEE Transactions on Software Engineering 34 (2008) 181–196.

[29] S. McIntosh, Y. Kamei, B. Adams, A. E. Hassan, The impact of code review coverage and code review participation on software quality: A case study of the QT, VTK, and ITK projects, in: Proceedings of the 11th Working Conference on Mining Software Repositories, 2014, pp. 192–201.

[30] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, M. W. Godfrey, Investigating code review quality: Do people and participation matter?, in: Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution, 2015, pp. 111–120.

[31] A. Mockus, Missing data in software engineering, in: Guide to Advanced Empirical Software Engineering, 2008, pp. 185–200.

[32] N. Nagappan, T. Ball, A. Zeller, Mining metrics to predict component failures, in: Proceedings of the 28th International Conference on Software Engineering, 2006, pp. 452–461.

[33] C. Ni, W. Wang, K. Yang, X. Xia, K. Liu, D. Lo, The best of both worlds: Integrating semantic features with expert features for defect prediction and localization, in: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2022, pp. 672–683.

[34] X. Chen, F. Xu, Y. Huang, N. Zhang, Z. Zheng, JIT-Smart: A multi-task learning framework for Just-In-Time defect prediction and localization, Proceedings of the ACM on Software Engineering 1 (2024) 1–23.

[35] K. Herzig, S. Just, A. Zeller, It's not a bug, it's a feature: How misclassification impacts bug prediction, in: 2013 35th international conference on software engineering (ICSE), IEEE, 2013, pp. 392–401.

[36] P. S. Kochhar, T.-D. B. Le, D. Lo, It's not a bug, it's a feature: does misclassification affect bug localization?, in: Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014, Association for Computing Machinery, New York, NY, USA, 2014, p. 296–299.