

Article

PreCrash: A Multi-Feature Fusion Approach for Predicting the Reproducibility of Crash Reports

Shaoting Liu, Haiyan Xu, Ziqi Shuai  and Jifeng Xuan * 

School of Computer Science, Wuhan University, Wuhan 430072, China; shaotingliu@whu.edu.cn (S.L.); haiyanxu@whu.edu.cn (H.X.); szq@whu.edu.cn (Z.S.)

* Correspondence: jxuan@whu.edu.cn; Tel.: +86-27-6877-6139

Abstract

Crash reports are critical for software debugging and maintenance. In software maintenance, developers reproduce a crash to locate the root cause via analyzing the crash report. A crash report is a document that records details of the crash as it occurred. The crash reports cannot be reproduced by developers because numerous crash reports in real-world issue-tracking systems contain incomplete reproduction steps and ambiguous descriptions. Existing works have explored the reproduction of crash reports for Android applications. These works focus on the process of reproducing crashes based on steps to reproduce and stack traces. However, it is a challenging task to manually identify reproducible crash reports from a large number of crash reports that consist of missing reproduction steps. To address this challenge, we propose PreCrash, a multi-feature fusion approach for predicting the reproducibility of a crash report. This approach helps identify reproducible crash reports before developers expend manual effort trying to reproduce the crashes. First, PreCrash automatically extracts 49 features from each crash report, which cover basic metadata, contextual content, comments, and supplementary evidence in crash reports. Second, PreCrash integrates four text representation models, including Word2Vec, GloVe, FastText, and BERT representations. Third, PreCrash employs a TextCNN-based prediction model to capture semantic patterns based on the integrated text representation. We evaluate PreCrash on 4133 crash reports collected from 102 real-world Java and Android projects on GitHub. Experimental results show that PreCrash achieves an accuracy of 0.91 and an F1-score of 0.91. PreCrash outperforms traditional machine learning models and single text representation baselines.

Keywords: crash reports; reproducibility prediction; multi-feature fusion; software maintenance; GitHub issues



Academic Editor: Shakeel Ahmad,
Ismet Gocer, Raza Hasan and
Zuhaib Khan

Received: 21 July 2026

Revised: 4 September 2026

Accepted: 9 September 2026

Published: 13 September 2026

Copyright: © 2026 by the authors.
Licensee MDPI, Basel, Switzerland.
This article is an open access article
distributed under the terms and
conditions of the [Creative Commons
Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

1. Introduction

Software crashes are disruptive failures in modern software systems [1]. A crash typically occurs when a program encounters an unrecoverable runtime error, like a null pointer dereference, a memory access out-of-bounds error, and a dependency conflict. Such a failure triggers the operating system and terminates the running process to protect system integrity [2]. A software crash can contain a crash symptom and a crash report [3]. A crash symptom is the real-time state of the software that the user can observe; a crash report is a structured description of this symptom [4]. A crash report consists of a description, steps to reproduce, a log, and developer comments. Figure 1 shows two examples of crash reports in Project Elasticsearch and Project Status with the description, steps to reproduce, log, and

developer comments. Thus, crash reports can serve as a vital basis for software debugging, maintenance, and quality assurance.

Crash reports help developers understand the actual scenarios of a crash during the software maintenance process [5]. A high-quality crash report can guide developers in identifying the root cause of a crash, reproducing the crash, and repairing the crashing code [6]. However, numerous reports contain ambiguous descriptions, incomplete steps, and insufficient context [7]. Developers need to spend an extensive amount of time and effort in determining whether a crash report is reproducible before developers start debugging. The reproducibility of a crash report can affect how efficiently a programmer can debug the crash [8]. If a developer can trigger the same crash in the specified environment based on the information in the report, the crash report is considered reproducible; otherwise, the report is non-reproducible. Non-reproducible crash reports may lack detailed steps or specific environmental details. The reproducibility of a crash report can help developers prioritize reports with high debugging value. This also supports problem classification, resource allocation, and automated software maintenance.

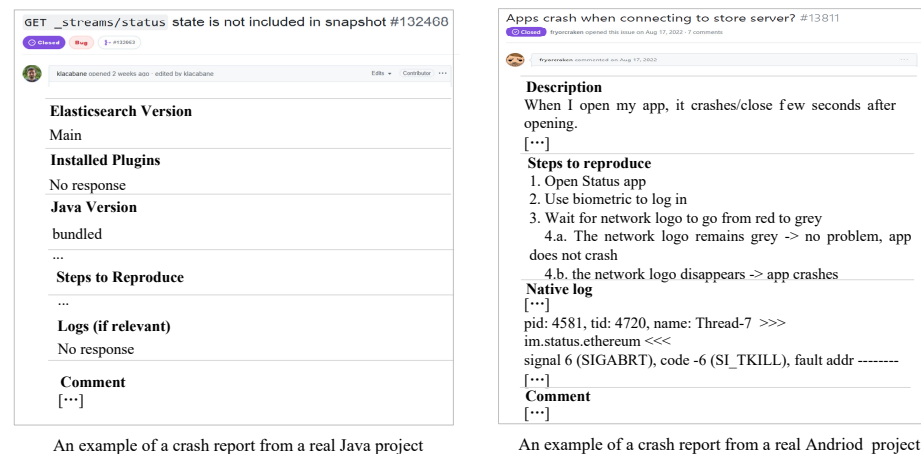


Figure 1. Two examples of crash reports from Project Elasticsearch and Project Status.

Existing studies [9–11] have explored several tasks related to crashes and defects, including crash report classification, duplicate defect report detection, defect prioritization, severity prediction, and fault localization. These studies indicate that textual details, stack traces, annotations, and project metadata can provide useful clues for software maintenance tasks. However, existing methods have not addressed the identification of the reproducibility of crash reports. These existing methods depend primarily on stack trace features to reproduce a crash report, while failing to adequately account for the interactions among report content, runtime environment, developer feedback, and supplementary evidence. Thus, these methods may fail to capture the multidimensional factors that determine whether a crash report can be reproduced in practice.

In this paper, we propose PreCrash, short for Predicting the reproducibility of Crash reports, a multi-feature fusion approach for predicting the reproducibility of crash reports. PreCrash considers the prediction of crash report reproducibility as a binary classification task, i.e., reproducible or not. For a given crash report, PreCrash extracts 49 features from each crash report, including basic metadata, contextual content, comments, interaction features, and supplementary evidence. These features are designed to capture both the static content of the report and the dynamic feedback generated during interactions between developers and users. In addition, PreCrash integrates multiple text representation methods to model the semantic information of crash reports. Specifically, PreCrash combines Word2Vec, GloVe, FastText, and BERT representations to capture crash report semantics

from different perspectives. Among these representations, Word2Vec and GloVe provide stable distributional semantic representations; FastText improves the representation of rare words, technical tokens, and domain-specific terms through subword information; BERT representations further provide contextual semantic information. After feature representation and fusion, PreCrash employs a TextCNN-based prediction model to capture local textual patterns and perform reproducibility classification.

We evaluate PreCrash on a dataset of 4133 crash reports, which are collected from 102 real-world open-source projects hosted on GitHub, covering the Java and Android software ecosystems. The dataset contains 2747 non-reproducible crash reports and 1386 reproducible crash reports. Experimental results demonstrate that PreCrash achieves an accuracy of 0.91 and an F1-score of 0.91. This outperforms traditional machine learning models and single text representation baselines. This experimental result suggests that multi-feature fusion can effectively improve reproducibility assessment. This study is inspired by the definition of the problem of crash report reproducibility and the concept of task adaptation. This study defines the prediction of crash report reproducibility as an independent software maintenance task and adapts existing representation methods to address this specific problem.

The main contributions of this paper are summarized as follows,

- We identify the problem of predicting reproducibility of crash reports as a binary classification task. Each crash report is classified as either reproducible or non-reproducible. To the best of our knowledge, this is the first work that addresses the prediction of reproducibility for crash reports.
- We propose a multi-feature fusion method, called PreCrash, which integrates text, context, interactions, and supplementary evidence to predict the reproducibility of crash reports.
- We construct a labeled dataset consisting of 4133 crash reports sourced from 102 real-world GitHub projects. Experimental results demonstrate that PreCrash outperforms other approaches in terms of accuracy, precision, recall, and F1 score.
- We implement PreCrash as an automated tool, which integrates Word2Vec, GloVe, FastText, and BERT representations within a TextCNN-based prediction framework. The prototype of PreCrash and experimental data are publicly available (see Section 4.5).

The remainder of this paper is organized as follows. Section 2 introduces the background of crash reports and the motivation of our work. Section 3 presents details of the proposed PreCrash approach. Section 4 describes the dataset collection process, baseline, evaluation metrics, implementation, and research questions of our study. Section 5 presents the experimental results of the proposed PreCrash approach. Section 7 reviews related works on crash report analysis and crash reproduction. Finally, Section 8 concludes our works.

2. Background and Motivation

2.1. Crash Reports

Crash reports are crucial clues for developers in identifying defects during software maintenance [12]. A crash report typically contains steps to reproduce the crash, runtime contextual information, exception types, and stack traces [13]. By analyzing crash reports, developers can locate the root causes of defects more efficiently. They can also assess the reproducibility of crash reports and verify whether repair solutions correctly fix the reported failures. Thus, crash report analysis plays an important role in improving software quality and improving the user experience. Depending on the issue-tracking system, the format and content of crash reports may vary. This paper focuses on the format and content

of crash reports collected from GitHub issues. In general, a crash report may contain the following contents:

- Basic data include the report title, application name, version number, device model, and operating system version.
- Environmental data include CPU usage, network status, permission settings, and other runtime configurations.
- Crash data include exception types, stack traces, crash time, and other failure-related details.
- Comment data include user descriptions of the issue, feedback on whether the crash can be reproduced, and repair suggestions.

The reproducibility of crash reports is closely related to debugging efficiency. For reproducible reports, developers can observe the failure symptoms, examine the program state, trace the execution path, and verify patches. For unreproducible reports, developers may need to infer missing information, request additional details from users, or spend substantial effort recreating the execution environment. This process increases maintenance costs and delays defect resolution. Thus, the prediction of the reproducibility of crash reports provides strong support for issue classification and debugging prioritization.

In real-world open-source projects, the reproducibility of a crash report is not determined by a single factor. Instead, the reproducibility is influenced by the interaction of multiple factors, including the content of the report, the runtime environment, user actions, the project context, and developer feedback. For example, a report may contain a detailed stack trace, it may still be difficult to reproduce without the required operating system, device model, or dependency versions. Similarly, a report containing only a brief textual description can become reproducible by including developer confirmation, temporary workarounds, or links to relevant commits in the discussion. This indicates that prediction of reproducibility in crash reports should take multiple factors into account, rather than relying solely on textual descriptions or stack traces.

2.2. Motivation

In modern software development, crashes are an unavoidable part of software systems. This is especially true for complex application software, where crashes are common events throughout the software development lifecycle. Software crashes usually include program exceptions, program errors, and unexpected application terminations [14]. These crashes not only interrupt program execution but may also damage user experience and system stability. Thus, how to effectively detect and handle crash issues has become a key problem for improving software quality and user satisfaction. Crash reports serve as an important data bridge between users and developers. They record key data when a crash occurs, including error messages, stack traces, system states, application versions, and configurations [15]. These reports provide developers with feedback from real software usage. Such feedback can help developers trace and fix the root causes of crashes.

In the case of open-source projects on GitHub, developers often face a large volume of backlogged crash reports [16]. However, many of these reports lack sufficient data to reproduce the issue. For example, a crash report might contain only a stack trace but fail to document the user's steps. Reports may also lack detailed descriptions of specific software versions or runtime environments. Rahman et al. [17] analyze 576 crash reports and identify 12 key factors that prevent crash reports from being reproduced. Among these factors, duplicate crashes, random crashes, missing information, and ambiguous descriptions are among the key factors that prevent developers from reproducing crash reports. These factors delay software iterations and defect fixes, and they may also result in critical bugs

remaining unresolved for extended periods, potentially affecting software stability and user experience.

This motivates our work. We propose the PreCrash approach, which integrates 49 features extracted from crash reports, including stack traces, user behavior, system environment, and operating system versions. Based on the correlations among these features, the PreCrash method can predict the reproducibility of crash reports. Specifically, this approach introduces deep learning and multi-feature fusion techniques to extract latent features from crash reports, improving the accuracy and efficiency of reproducibility of crash reports.

3. Proposed Approach

This section presents the proposed approach PreCrash for predicting the reproducibility of crash reports. PreCrash treats the prediction of crash report reproducibility as a binary classification task. The goal of this method is to predict whether a given crash report is reproducible. To achieve this, PreCrash integrates multiple information sources, including text descriptions, runtime environments, developer and user interactions, and supplementary evidence. The overall workflow consists of four main stages: feature extraction, text representation, multi-feature fusion, and TextCNN-based prediction.

3.1. Overview of PreCrash

Figure 2 shows the overview of the PreCrash approach. PreCrash takes crash reports from the issue-tracking system as input; the output is the reproducibility of each crash report. First, PreCrash processes collected crash reports by filtering, cleaning, and labeling them. Second, PreCrash extracts 49 features from each crash report to characterize the report from multiple dimensions, including basic metadata, contextual information, runtime environment, interactions, and supplementary evidence. The extracted feature information is normalized and serialized into a unified textual input, ensuring that heterogeneous information from different feature dimensions can be represented in a consistent form. Third, the unified textual input is independently encoded using four text representation methods, including Word2Vec [18], GloVe [19], FastText [20], and BERT [21], to capture complementary information from different representation perspectives. Since these representations may have different dimensions, they are projected into a common dimensional space and concatenated to form a fused multi-source representation. Fourth, the fused representation is used as the input to TextCNN, which extracts high-level discriminative patterns through convolution and pooling operations and outputs the reproducibility probability of the crash report.

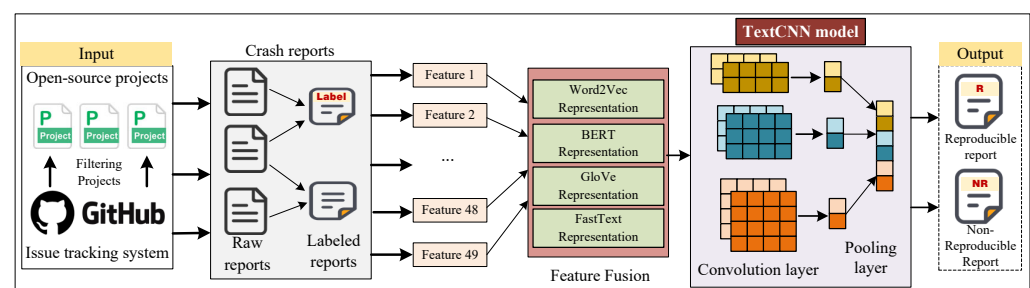


Figure 2. Overview of the proposed approach PreCrash.

Formally, given a crash report r_i , PreCrash predicts a label y_i ,

$$y_i = \begin{cases} 1, & \text{if } r_i \text{ is reproducible,} \\ 0, & \text{if } r_i \text{ is non-reproducible.} \end{cases} \quad (1)$$

The prediction function can be defined as

$$\hat{y}_i = F(r_i; \Theta), \quad (2)$$

where $F(\cdot)$ denotes the PreCrash model, Θ represents the learnable parameters, and $\hat{y}_i$ is the predicted probability that the crash report is reproducible.

3.2. Feature Extraction

To characterize the factors that affect the reproducibility of crash reports, we designed 49 features based on existing studies [17,22,23] and our empirical analysis of crash reports. The reproducibility of crash reports is affected by multiple categories of data. Thus, PreCrash extracts a set of multi-dimensional features from each crash report. These features are categorized into four groups: basic metadata features, situational and environmental features, comment and interaction features, and supplementary evidence features. Table 1 summarizes the main feature categories used in PreCrash.

To avoid directly encoding the annotation outcome into the model input, features related to reproduction discussions are defined to capture the presence and content of reproduction-related information rather than the final reproduction outcome. In particular, these features do not explicitly encode whether a reproduction attempt succeeded or failed. Two reasons for selecting these features are as follows: (1) We reviewed existing research on crash reproduction, bug reports, runtime environments, and issue discussions to identify candidate factors that might be related to the reproducibility of crash reports. (2) We analyzed real crash reports to identify additional information that frequently appears in GitHub issues and might provide useful evidence for reproducibility assessments.

Basic metadata features. Basic metadata fields describe general details of the crash report. These fields include the repository name, issue ID, title, creation time, report lifecycle, labels, number of comments, milestones, assignees, author roles, and last update time. These fields reflect the administrative context of the report. For example, a report with many comments may indicate that developers have invested considerable effort in discussing or verifying the issue. A long lifecycle may indicate that the crash issue is complex or difficult to reproduce [24].

Contextual and environmental features. Contextual and environmental characteristics describe the runtime conditions at the time of a crash. These characteristics include the operating system type and version, hardware configuration, device model, dependency versions, build tool versions, framework versions, network conditions, stack trace, exception type, recurrence frequency, user interaction patterns, crash severity, input data, expected behavior, and actual behavior [25]. A crash may occur only under specific operating systems, devices, library versions, network conditions, or execution configurations.

Comment and interaction features. Comment and interaction features capture additional contextual information provided during discussions of crash reports. These features include developer debugging suggestions, clarification or refinement of reproduction steps, temporary fixes, runtime environment information, code change references, environment variables, and version-specific reproduction context [2]. These features describe reproduction-related discussions without explicitly encoding whether a reproduction attempt succeeded or failed.

Supplementary evidence features. Supplementary evidence features describe the additional information attached to crash reports. These data include logs, screenshots, videos,

input/output examples, external links, configuration files, command-line arguments, code snippets, project versions, and the clarity of the steps to reproduce the issue. These features reflect the completeness and readability of the report [26]. Crash reports with complete supplementary evidence are generally easier for developers to understand and reproduce.

Table 1. Summary of the 49 features used for crash report reproducibility prediction. The features are grouped into four categories according to their primary information sources and functional roles.

No.	Feature Name	Description
Basic Metadata Features		
1	Repository full name	The full name of the repository to which the crash report belongs.
2	Issue number [17]	The unique identifier of the crash report.
3	Report title [17]	A brief description of the crash.
4	Creation time	The timestamp when the crash report was created.
5	Lifecycle [17]	The time interval from report creation to report closure.
6	Labels [22]	The label information associated with the crash report.
7	Number of comments [17]	The number of comments in the report, reflecting community participation.
8	Milestone association	Whether the report is associated with a specific development milestone.
9	Assignee [17]	The developer assigned to handle the crash report.
10	Author role	The role of the report author in the repository.
11	Last update time	The timestamp when the report was last updated.
12	Repository maintenance	Whether the repository is actively maintained.
Contextual and Environmental Features		
13	System type	The type of operating system mentioned in the report.
14	System version	The operating system version mentioned in the report.
15	Hardware configuration	The hardware environment information involved in the report.
16	Device model	The device model on which the crash occurred.
17	Hardware requirement	Whether specific hardware is required to reproduce the issue.
18	JDK/JRE version	Whether the crash depends on a specific JDK/JRE version.
19	Build tool [23]	Whether the crash depends on a specific build tool version.
20	Specific library	Whether the crash depends on a specific third-party library version.
21	External testing framework [23]	Whether the crash depends on a specific external testing framework version.
22	Network condition	Whether the crash is related to network conditions.
23	Stack trace [22]	Whether the report contains complete stack trace information.
24	Exception type	The type of exception involved in the crash.
25	Reproduction frequency	The frequency with which the crash occurs.
26	User interaction mode	The user interaction pattern when the crash occurs.
27	Crash severity [17]	The impact level of the crash on users and the system.
28	Input data	The input data information involved in the report.
29	Expected behavior [22]	The expected behavior described in the report.
30	Actual behavior [22]	The actual behavior observed when the crash occurs.
Comment and Interaction Features		
31	Debugging suggestions	Whether comments contain debugging suggestions from maintainers or contributors.
32	Reproduction-step discussion	Whether comments contain additional clarification, refinement, or procedural information related to the reproduction steps.
33	Temporary fix [17]	Whether comments contain temporary fixes or patches.
34	Reproduction feedback	Whether comments contain information describing reproduction attempts or reproduction-related investigation, without encoding the outcome of these attempts.
35	Runtime environment [23]	Whether comments contain specific runtime environment information.
36	Code change reference	Whether comments refer to relevant code changes.
37	Environment variables [23]	Whether comments contain environment variables or startup parameters.
38	Specific version	Whether comments contain version-specific information associated with the reproduction context of the crash, without encoding the reproduction outcome.
Supplementary Evidence Features		
39	Runtime logs [23]	Whether runtime logs or stack trace information are provided.
40	Input/output examples [23]	Whether input and output examples are provided.
41	External dependencies [17]	Whether external dependencies are explicitly specified in the report.
42	Project version [23]	Whether a specific project version is included.
43	External resources	Whether the report contains links to external resources.
44	Screenshots or videos [22]	Whether screenshots or videos are attached as supporting evidence.
45	Configuration files [22]	Whether specific configuration files or runtime parameters are included.
46	Report description [17]	Whether the overall report description is clear and understandable.
47	Code snippets	Whether code snippets are attached to the report.
48	Steps to reproduce [17]	Whether the report contains an explicit sequence of steps related to triggering the crash, without assessing whether the steps are complete or ultimately successful.
49	Number of affected users	The number of affected users mentioned in the report.

After feature extraction, each crash report r_i is represented as a feature vector:

$$\mathbf{s}_i = [s_{i1}, s_{i2}, \dots, s_{im}], \quad (3)$$

where m denotes the number of structured features. In this study, PreCrash extracts 49 features from each crash report. These structured features are later combined with textual representations to form the final input of the prediction model.

3.3. Text Representation and Multi-Feature Fusion

Crash reports contain titles, descriptions, stack traces, reproduction steps, runtime environments, comments, and other report-related information. For each crash report, PreCrash first extracts the 49 features described in Section 3.2. These extracted features characterize the crash report from multiple dimensions, including basic metadata, contextual information, runtime environments, interactions, and supplementary evidence. The extracted feature information is then normalized and serialized into a unified textual input. Thus, the 49 features are incorporated into the model before text representation rather than being concatenated with the output of TextCNN as an additional feature vector.

For each crash report, the unified textual input is tokenized into a sequence of words:

$$\mathbf{w} = (w_1, w_2, \dots, w_L), \quad (4)$$

where w_i denotes the i -th token and L denotes the sequence length.

Unlike general natural language, crash reports contain numerous domain-specific terms, including API names, class names, file paths, version numbers, exception messages, and stack traces. A single text representation approach may be insufficient to capture all these patterns. Thus, PreCrash employs four supplementary text representation techniques: Word2Vec, GloVe, FastText, and BERT representations. Specifically, Word2Vec captures semantic context through word vectors learned from local contexts; GloVe captures global co-occurrence statistics and provides stable semantic representations; FastText captures subword-level information and is particularly useful for rare technical terms, spelling variants, and domain-specific identifiers; and BERT captures contextual semantics and can generate different representations for the same token according to its surrounding context.

The four representation methods independently encode the same unified textual input generated from the 49 extracted features. PreCrash represents the same crash report from four complementary semantic perspectives before performing representation fusion.

For each token w_i , the four representation methods generate four embedding vectors:

$$\mathbf{e}_i^{w2v}, \quad \mathbf{e}_i^{glv}, \quad \mathbf{e}_i^{ft}, \quad \mathbf{e}_i^{bert}. \quad (5)$$

Since these embedding vectors may have different dimensions, PreCrash maps them to a uniform dimension using linear projection:

$$\tilde{\mathbf{e}}_i^{(k)} = \mathbf{P}^{(k)} \mathbf{e}_i^{(k)} + \mathbf{b}^{(k)}, \quad k \in \{w2v, glv, ft, bert\}, \quad (6)$$

where $\mathbf{P}^{(k)}$ is the projection matrix, $\mathbf{b}^{(k)}$ is the bias vector, and k denotes the embedding type.

After projection, the four representations of each token are concatenated to obtain a multi-source token representation:

$$\mathbf{e}_i = [\tilde{\mathbf{e}}_i^{w2v}; \tilde{\mathbf{e}}_i^{glv}; \tilde{\mathbf{e}}_i^{ft}; \tilde{\mathbf{e}}_i^{bert}]. \quad (7)$$

For the entire serialized crash report sequence, the fused representation matrix is constructed as follows:

$$\mathbf{X} = \begin{bmatrix} \mathbf{e}_1 \\ \mathbf{e}_2 \\ \vdots \\ \mathbf{e}_L \end{bmatrix}. \quad (8)$$

The fused matrix $\mathbf{X}$ contains complementary information captured by Word2Vec, GloVe, FastText, and BERT from the same feature-enriched crash report input. It is subsequently used as the input to TextCNN. The convolutional and pooling layers of TextCNN further extract high-level discriminative patterns from the fused representation, and the resulting representation is used to predict the reproducibility probability of the crash report.

3.4. Model Construction

The core objective of our research is to use the resulting representations of crash reports to predict crash report reproducibility. To achieve this goal, the model development process consists of three main components: feature-based input construction, multi-source text representation and fusion, and TextCNN-based prediction. Figure 3 illustrates the overall construction process of the proposed prediction model. For each crash report, PreCrash first extracts the predefined 49 features and normalizes and serializes the extracted information into a unified textual input. The same input is then independently encoded using Word2Vec, GloVe, FastText, and BERT to obtain four complementary representations. The fused representation is subsequently fed into TextCNN. Multiple convolution kernels are used to extract local semantic patterns at different granularities, followed by ReLU activation and max pooling to obtain salient features. A dropout layer is then applied to reduce overfitting, and the resulting representation is passed through a fully connected layer and a sigmoid function to estimate the reproducibility probability of the crash report.

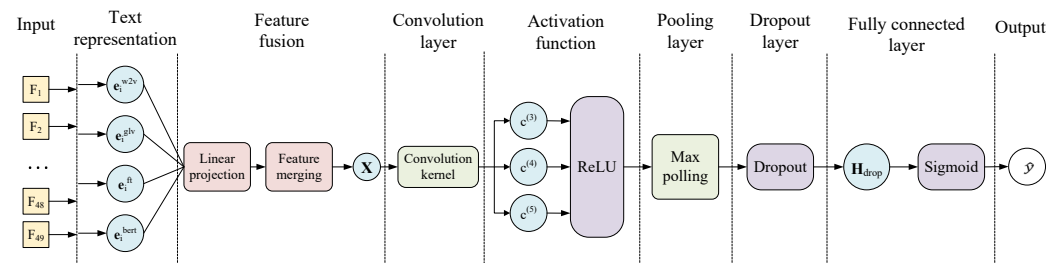


Figure 3. Overall construction process of the proposed prediction model.

Text representation. Text representation refers to the process of converting raw natural language text into a vector format that computers can process. Crash reports typically include a title, description, and notes. They contain highly technical details like domain-specific terminology, library versions, operating systems, and stack traces. Such text enters a model directly, the model may be affected by noise and redundant information. Thus, the text must be standardized before feature engineering.

First, invalid symbols are removed from the original crash reports, and stop words are deleted to improve the consistency of the corpus. Then, the text is serialized into a word sequence of length L :

$$\mathbf{w} = (w_1, w_2, \dots, w_L) \quad (9)$$

where w_i denotes the i -th word, and L denotes the actual length of the text. This serialization preserves the word order, allowing the model to identify contextual dependencies among different words. In descriptions of crash reproduction steps, word order can directly affect semantic logic. It is also a necessary prerequisite for subsequent vectorization and convolution operations.

Then, PreCrash maps the above text sequence into a continuous vector space. PreCrash combines four embedding methods, namely, Word2Vec, GloVe, FastText, and BERT, to construct feature representations and enhance representation capability.

Word2Vec representation. Word2Vec is a static word embedding method based on context prediction [27]. PreCrash uses the Continuous Bag of Words (CBOW) architecture [28] to train word vectors. The core idea of Word2Vec is that words with similar contexts tend to have similar meanings. This representation is suitable for capturing local distributional information in the corpus. The embedding matrix can be defined as follows:

$$E_{w2v} \in \mathbb{R}^{|V| \times d_w} \quad (10)$$

where $|V|$ denotes the vocabulary size and d_w denotes the dimension of the word vector. Given the one-hot representation $\mathbf{o}_i \in \mathbb{R}^{|V|}$ of word w_i , its corresponding word vector is as follows:

$$\mathbf{e}_i^{w2v} = E_{w2v}^\top \mathbf{o}_i \in \mathbb{R}^{d_w} \quad (11)$$

This indicates that the word w_i is mapped into a low-dimensional dense vector through the lookup table E_{w2v} . Word2Vec is adopted because it is computationally efficient, fast to train, and capable of capturing local contextual relationships among words.

GloVe representation. GloVe (Global Vectors for Word Representation) is a word embedding method based on global word statistics [29]. PreCrash constructs a word co-occurrence matrix and factorizes it to obtain word vectors. The core idea of GloVe is that the meaning of a word can be reflected by its relationships with other words. Therefore, it is suitable for capturing global semantic information in the corpus. The embedding matrix can be represented as follows:

$$E_{glv} \in \mathbb{R}^{|V| \times d_g} \quad (12)$$

where d_g denotes the word vector dimension. The vector representation of word w_i is as follows:

$$\mathbf{e}_i^{glv} = E_{glv}^\top \mathbf{o}_i \in \mathbb{R}^{d_g} \quad (13)$$

where E_{glv} is learned by minimizing the error between word-pair co-occurrence probabilities, so that global semantic relationships can be better captured. GloVe is adopted because it effectively uses global statistical information and captures latent relationships among words.

FastText representation. FastText is a word embedding method based on the bag-of-words model [20]. It can generate word representations using n -gram features. PreCrash divides each word into n -grams and embeds each n -gram to obtain the final word vector. The core idea of FastText is that the meaning of a word depends not only on the word itself but also on its internal structure. Thus, it is suitable for capturing morphological information in the corpus. The mapping function can be defined as follows:

$$\mathbf{e}_i^{ft} = \frac{1}{|G(w_i)|} \sum_{g \in G(w_i)} \mathbf{z}_g \in \mathbb{R}^{d_f} \quad (14)$$

where $G(w_i)$ denotes the set of subwords of word w_i , $\mathbf{z}_g$ denotes the embedding of subword g , and $|G(w_i)|$ denotes the number of subwords. Since crash reports frequently contain special function names, customized variable names, technical terms, spelling variations, and unseen words, FastText can still generate effective word vectors for rare tokens and improve model robustness.

BERT representation. BERT (Bidirectional Encoder Representations from Transformers) is a pretrained language model based on the Transformer architecture [30]. PreCrash tokenizes the input text and generates context-aware embeddings for each token. The

core idea of BERT is that the meaning of a word depends on its context. Therefore, BERT is suitable for capturing deep semantic information in crash report text. The mapping function can be represented as follows:

$$\mathbf{e}_i^{\text{bert}} = h_i \in \mathbb{R}^{d_b} \quad (15)$$

where d_b denotes the hidden dimension of BERT. Unlike static embeddings, BERT can generate different representations for the same word in different contexts, thereby addressing the problem of polysemy. BERT is adopted because it can effectively model contextual information and improve the semantic representation capability of word vectors.

Since the above embeddings have different dimensions, a linear projection is used to map them into a unified dimension d :

$$\tilde{\mathbf{e}}_i^{(k)} = \mathbf{P}^{(k)} \mathbf{e}_i^{(k)} + \mathbf{b}^{(k)}, \quad k \in \{\text{w2v, glv, ft, bert}\} \quad (16)$$

where $\mathbf{P}^{(k)} \in \mathbb{R}^{d \times d_k}$ is the projection matrix, $\mathbf{b}^{(k)} \in \mathbb{R}^d$ is the bias vector, and d_k denotes the original dimension of each embedding. The projected embeddings are concatenated to obtain the multi-source embedding:

$$\mathbf{e}_i = [\tilde{\mathbf{e}}_i^{\text{w2v}}; \tilde{\mathbf{e}}_i^{\text{glv}}; \tilde{\mathbf{e}}_i^{\text{ft}}; \tilde{\mathbf{e}}_i^{\text{bert}}] \in \mathbb{R}^{4d} \quad (17)$$

The representation of the whole text is as follows:

$$\mathbf{X} = \begin{bmatrix} \mathbf{e}_1 \\ \mathbf{e}_2 \\ \vdots \\ \mathbf{e}_L \end{bmatrix} \in \mathbb{R}^{L \times 4d} \quad (18)$$

The advantage of this representation is that it combines the stability of static embeddings and the contextual sensitivity of dynamic embeddings. It captures the semantic information of crash reports from multiple perspectives and provides stronger input features for subsequent modeling.

For text preprocessing, the crash report text is tokenized and converted into token sequences, which are padded or truncated to the predefined maximum sequence length. The tokenizer is fitted only on the training subset. Word2Vec and FastText representations are also trained exclusively on the training subset, and the held-out data are transformed using the representations learned from the training data. GloVe embeddings are obtained from an external pretrained GloVe model and are not retrained on the crash report dataset. For the Transformer-based representation, we use the pretrained `albert-base-v2` checkpoint as a fixed feature extractor (user guide for the `albert-base-v2` model, <https://huggingface.co/albert/albert-base-v2>, accessed on 11 March 2026). The pretrained Transformer model is used as a fixed feature extractor, and its parameters are not updated during feature extraction. In addition, corpus-dependent representations such as Word2Vec and FastText are trained only on the training subset, without using the validation or testing data.

Based on the feature representation, we train a TextCNN model for prediction. Unlike recurrent neural networks, TextCNN offers high computational efficiency, a simple architecture, and a strong ability to capture local patterns. This model is suitable for classification tasks involving short and medium-length texts. The TextCNN model adopted in this paper consists primarily of convolutional and pooling layers.

Convolutional layer. The convolutional layer uses multiple convolution kernels with different window sizes to extract n -gram features. The calculation is defined as follows:

$$c_i^{(h,m)} = f\left(\langle \mathbf{W}^{(h,m)}, \mathbf{X}_{i:i+h-1} \rangle + b^{(h,m)}\right) \quad (19)$$

where $\mathbf{X}_{i:i+h-1} \in \mathbb{R}^{h \times 4d}$ denotes a submatrix with length h , $\mathbf{W}^{(h,m)}$ denotes the convolution kernel weight, $b^{(h,m)}$ denotes the bias term, and $f(\cdot)$ denotes the ReLU activation function. This operation can automatically identify key triggering patterns such as “click crash” or “launch failure”.

Pooling layer. After the convolutional layer, temporal max pooling is applied to the generated feature map:

$$\hat{c}^{(h,m)} = \max_i c_i^{(h,m)} \quad (20)$$

The advantage of pooling is that it automatically selects the most salient local patterns and ignores irrelevant information to improve the robustness of the model.

After convolution and pooling, the features extracted by convolution kernels of different sizes are aggregated into several fixed-length vectors. These vectors represent the most semantic patterns of the crash report at different scales. For example, they may represent triggering crashes features at the trigram level. To obtain a comprehensive sentence-level representation, these features from different sources are concatenated into a high-dimensional vector. This vector provides a fused representation of important features from multiple scales and multiple channels. Then, a Dropout layer is introduced to reduce the risk of overfitting and improve generalization. Finally, the concatenated feature vector is fed into a fully connected layer. Through linear transformation and nonlinear activation, the complex feature space is mapped into a probability distribution. The probability indicates whether the report belongs to the reproducible or non-reproducible class, thereby completing the classification prediction.

Loss function and optimization. This paper adopts binary cross-entropy as the loss function:

$$\mathcal{L} = -\frac{1}{N} \sum_{n=1}^N [y_n \log \hat{y}_n + (1 - y_n) \log(1 - \hat{y}_n)] \quad (21)$$

where N denotes the number of training samples, $y_n \in \{0, 1\}$ denotes the ground-truth label, and $\hat{y}_n$ denotes the predicted value of the model.

Decision results. The model determines the final label according to the predicted probability and threshold τ :

$$label = \begin{cases} R, & \hat{y} \geq \tau, \\ NR, & \hat{y} < \tau, \end{cases} \quad (22)$$

where R denotes reproducible and NR denotes non-reproducible. In this paper, the threshold is set to $\tau = 0.5$.

4. Experimental Setup

In this section, we describe the experimental setup for evaluating the performance of PreCrash in the prediction of the reproducibility of crash reports.

4.1. Dataset Collection

To construct a prediction model for crash report reproducibility, we collected data from 102 real-world open-source projects on GitHub. The data collection process consists of the following steps: (1) Project selection. We use the number of GitHub stars as the ranking criterion and selected the Java and Android projects. Finally, we obtain

60 real-world Java projects and 42 real-world Android projects. (2) Crash report collection. We use the GitHub API to crawl all issues from the selected 102 projects (GitHub API documentation, <https://docs.github.com/en/rest>, accessed on 11 March 2026). From these issues, we extract titles, reproduction steps, comments, and related contextual information. (3) Feature extraction and cleaning. We remove reports that do not contain the minimum information required for extracting basic information features, contextual and environmental features, or comment and interaction features. After this initial cleaning step, we remain with 4595 crash reports. (4) Missing value removal. We further remove reports with 25 or more missing features. Since PreCrash extracts 49 predefined features from each crash report, missing 25 or more features means that more than half of the feature information is unavailable. Thus, we use 25 missing features as a practical completeness threshold to ensure that the retained reports contain sufficient information for feature representation and prediction.

Based on this criterion, 462 reports are removed, leaving 4133 crash reports for manual annotation and subsequent experiments. Among the final 4133 reports, 2747 are labeled as non-reproducible (NR) and 1386 as reproducible (R). Since manual labeling is performed after data cleaning, the 462 removed reports do not have R/NR labels. Incomplete reports may represent difficult reproducibility cases, which may introduce selection bias. Table 2 presents the dataset used for crash report reproducibility prediction. As shown in Table 2, the dataset covers 102 real-world open-source projects. Among them, Java projects account for 58.8%, while Android projects account for 41.2%. This distribution ensures cross-platform diversity and improves the representativeness of the dataset. The dataset initially contained 4595 raw crash reports. After data cleaning, 4133 complete reports were retained, indicating that the dataset has relatively high data quality. This dataset can support statistical analysis as well as the validation and optimization of deep learning models.

Table 2. Summary of the dataset for crash report reproducibility prediction

Module	Total	Name	Number	Proportion
Projects	102	Java projects	60	58.8%
		Android projects	42	41.2%
Raw reports	4595	Complete reports	4133	89.9%
		Incomplete reports	462	10.1%
Labeled reports	4133	Non-reproducible (NR)	2747	66.5%
		Reproducible (R)	1386	33.5%

4.2. Dataset Labeling

To construct the reference labels for the prediction of crash report reproducibility, we manually assess all 4133 crash reports in the dataset. The labeling process is conducted by three authors and took about one month. Two authors independently reviewed each crash report and assigned it *reproducible* (R) or *non-reproducible* (NR). When the two annotators disagreed on a report, the third author independently reviewed the report and adjudicated the final label. The labeling process was based on a holistic assessment of the evidence available in each crash report rather than on a deterministic mapping from any single report field or any individual input feature. In particular, the presence of reproduction steps, a stack trace, runtime logs, or a reproduction-related comment was not by itself sufficient to determine the final label. Instead, the annotators jointly considered the completeness, specificity, consistency, and technical support of the available evidence.

A crash report is labeled as *reproducible* when the available evidence, considered as a whole, provided sufficient support that the reported crash could reasonably be triggered again under the described conditions. Such evidence could include (1) sufficiently

detailed reproduction procedures together with the corresponding environmental or runtime information; (2) technical runtime evidence, such as stack traces, execution logs, input/output examples, or configuration information that was consistent with the reported failure; and (3) reproduction-related discussions from developers or users that corroborated the reported behavior under a clearly specified environment or version. Conversely, a crash report is labeled as *non-reproducible* when the available evidence is insufficient, inconsistent, or provided substantial evidence against reproduction. Typical cases included reports with missing or ambiguous reproduction procedures, insufficient input or configuration information, unspecified dependency or runtime conditions, or reproduction-related discussions indicating that the reported failure could not be reproduced under comparable conditions.

During annotation, conflicting evidence is also considered explicitly. For example, one participant may report that a crash could be reproduced, while another participant may report the opposite result. In such cases, the annotators considered whether the reported reproduction attempts are performed under comparable software versions, runtime environments, inputs, and dependency conditions. Technical evidence with greater specificity and verifiability, such as detailed execution logs, stack traces, input files, configuration information, or reproducible scripts, is given greater weight than an unsupported statement of reproduction success or failure. If the two primary annotators still reached different conclusions after considering the available evidence, the third author reviewed the report and determined the final label.

The final dataset contains two manually assessed reproducibility labels:

$$Y = \{R, NR\}, \quad (23)$$

where R denotes a report assessed as reproducible and NR denotes a report assessed as non-reproducible. These manually assessed labels are used as the reference labels for model training and evaluation.

4.3. Baselines

To the best of our knowledge, this is the first work to predict the reproducibility of crash reports. Then, to evaluate the effectiveness of PreCrash, we compare it with both traditional machine learning models and deep learning representation baselines. The traditional machine learning baselines are used to evaluate whether PreCrash outperforms classical classifiers based on structured features. The deep learning and text representation baselines are used to evaluate whether the multi-feature fusion strategy is more effective than using a single text representation.

The traditional machine learning baselines include Logistic Regression, Random Forest, Support Vector Machine, Decision Tree, and XGBoost.

- Logistic Regression (LR) is a linear classification model [31]. It estimates the probability that a crash report belongs to the reproducible class by learning a linear decision boundary. LR is used as a simple and interpretable baseline.
- Random Forest (RF) is an ensemble learning method based on multiple decision trees [31]. It combines the predictions of different trees to improve robustness and reduce overfitting. RF can capture nonlinear relationships among crash report features.
- Support Vector Machine (SVM) is a margin-based classifier [31]. It searches for an optimal hyperplane that separates reproducible and non-reproducible reports. SVM is suitable for high-dimensional feature spaces.
- Decision Tree (DT) is a tree-based classification model [32]. It recursively splits the feature space according to feature values. DT provides an interpretable baseline for understanding rule-based classification behavior.

- XGBoost is a gradient boosting model based on decision trees [33]. It builds an ensemble of weak learners and optimizes them sequentially. XGBoost is widely used in classification tasks because of its strong predictive performance.

The deep learning and text representation baselines include Word2Vec, GloVe, FastText, and BERT. These baselines are used to examine the effect of different textual representations on crash report reproducibility prediction.

- Word2Vec learns word embeddings from local context information [27]. It captures distributional semantic relationships between words and provides dense vector representations for crash report text.
- GloVe learns word embeddings from global word co-occurrence statistics [29]. It captures corpus-level semantic relationships and provides stable word representations.
- FastText extends word embedding learning by modeling subword information [20]. It is useful for handling technical tokens, rare words, function names, and identifiers that frequently appear in crash reports.
- BERT is a Transformer-based pretrained language model [21]. It provides contextual token representations and can capture semantic dependencies in natural language descriptions and comments.

These baseline models are evaluated as independent crash report reproducibility predictors and do not employ the multi-representation fusion mechanism of PreCrash. This comparison is intended to show the overall predictive performance of PreCrash with representative machine learning and single-representation baselines, rather than to isolate the contribution of the fusion mechanism. Compared with these baselines, PreCrash integrates multiple text representations together with crash report features to form a unified prediction model. The specific contribution of individual text representations and their fusion is further investigated through the controlled ablation experiments in Section 5.3. Together, these two comparisons evaluate both the overall effectiveness of PreCrash and the contribution of its multi-representation fusion strategy.

4.4. Evaluation Metrics

To effectively evaluate the performance of the model in the task of crash report reproducibility prediction, this paper uses four evaluation metrics: Accuracy, Precision, Recall, and F1-score [34]. The four metrics are defined as follows:

- TP (True Positive): Samples that are predicted as positive by the model and whose true labels are also positive. In this study, this refers to reproducible reports that are correctly identified as reproducible.
- TN (True Negative): Samples that are predicted as negative by the model and whose true labels are also negative. In this study, this refers to non-reproducible reports that are correctly identified as non-reproducible.
- FP (False Positive): Samples that are predicted as positive by the model but whose true labels are negative. In this study, this means that non-reproducible reports are incorrectly classified as reproducible.
- FN (False Negative): Samples that are predicted as negative by the model but whose true labels are positive. In this study, this means that reproducible reports are incorrectly classified as non-reproducible.

Accuracy measures the proportion of correctly predicted samples among all samples, and it is defined as follows:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \quad (24)$$

This metric reflects the overall classification performance of the model, but it may be biased when the class distribution is imbalanced.

Precision measures the proportion of true positive samples among all samples predicted as positive:

$$Precision = \frac{TP}{TP + FP} \quad (25)$$

This metric evaluates the reliability of the positive predictions, that is, how many reports predicted as reproducible are truly reproducible.

Recall measures the proportion of actual positive samples that are correctly identified by the model:

$$Recall = \frac{TP}{TP + FN} \quad (26)$$

This metric reflects the model's ability to cover positive samples, that is, how many reproducible reports are correctly identified among all reproducible reports.

To balance Precision and Recall, this paper further uses F1-score as a comprehensive metric:

$$F1\text{-score} = \frac{2 \times Precision \times Recall}{Precision + Recall} \quad (27)$$

F1-score is the harmonic mean of Precision and Recall. It is more suitable for evaluating model performance in scenarios with imbalanced class distributions. In summary, this paper evaluates the model using Accuracy, Precision, Recall, and F1-score. These metrics provide a comprehensive assessment from both overall classification performance and minority-class identification capability.

In addition, this paper uses three ranking metrics, including Recall@*k*, MRR, and MAP [35], to further evaluate the ranking-oriented performance of different text representations in crash report reproducibility prediction. The definitions of these metrics are as follows:

- Recall@*k*: Recall@*k* measures the proportion of relevant items retrieved within the top *k* results. It is calculated as follows:

$$Recall@k = \frac{\text{Number of relevant results retrieved in the top } k}{\text{Total number of relevant results}} \quad (28)$$

- MRR (Mean Reciprocal Rank): MRR calculates the average reciprocal rank of the first relevant result across all queries. It is used to evaluate how early the first relevant result appears in the ranked list. It is calculated as follows:

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{rank_i} \quad (29)$$

where $|Q|$ denotes the total number of queries, and $rank_i$ denotes the rank position of the first relevant result for the *i*-th query.

- MAP (Mean Average Precision): MAP calculates the average precision for each query and then averages the values over all queries. It reflects both ranking quality and overall retrieval effectiveness. It is calculated as follows:

$$MAP = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{m_i} \sum_{k=1}^n P_i(k) \cdot rel_i(k) \quad (30)$$

where m_i denotes the total number of relevant results for the *i*-th query, $P_i(k)$ denotes the precision at the top *k* results, and $rel_i(k)$ is a relevance indicator function, which equals 1 if the result at rank *k* is relevant and 0 otherwise.

4.5. Implementation

We have publicly released the prototype of our approach PreCrash and experimental data (prototype of PreCrash and experimental data, <https://github.com/lusaty/PreCrash>, accessed on 8 September 2026).

This study is conducted on Ubuntu 22.04.5 with Linux kernel version 5.15.0. The experimental platform is equipped with an Intel Xeon Gold 5222 processor, with a maximum frequency of 3.9 GHz. This system provides 8 logical cores and supports Hyper-Threading, which enables efficient execution for multitasking and deep learning training. For model implementation and parameter settings, this study uses a Keras-based TextCNN model to predict crash report reproducibility (Keras API documentation, <https://keras.io/api/>, accessed on 11 March 2026). For TextCNN, the maximum sequence length is set to 100 and the vocabulary size is limited to 60,000. The embedding dimension is 100, and the convolutional layer contains 128 filters with a kernel size of 5 and ReLU activation, followed by global max pooling and a fully connected layer with 10 units. A dropout rate of 0.2 is used to reduce overfitting. The model is trained for 5 epochs with a batch size of 32 using the Adam optimizer with a learning rate of 0.001 and binary cross-entropy loss. No early-stopping strategy is used because a fixed training schedule of five epochs is adopted. The tokenizer is fitted only on the training data, and all sequences are padded or truncated to the predefined maximum length. A validation set is used during training to monitor model performance. These settings allow the experiment to achieve good generalization performance while maintaining computational efficiency.

4.6. Research Questions

To investigate the effectiveness and feasibility of PreCrash, we design the following four Research Questions (RQs) from the perspectives of model performance, semantic representation, feature contribution, and multi-feature fusion.

- RQ1: How does PreCrash perform in the prediction of the reproducibility of crash reports?
This research question aims to evaluate the effectiveness of the proposed PreCrash approach in the prediction of the reproducibility of crash reports. Based on the constructed dataset, this RQ uses Accuracy, Precision, Recall, and F1-score to evaluate the prediction performance of PreCrash.
- RQ2: How effective are the four text representation models in capturing semantic information from crash reports?
This research question focuses on comparing different text representation methods, including Word2Vec, GloVe, FastText, and BERT, in terms of their ability to capture semantic information from crash reports. The goal is to investigate which vector representation method can accurately reflect the semantic relationships in crash reports and provide reliable input for subsequent feature modeling and prediction.
- RQ3: Which features play important roles in the proposed PreCrash?
This research question aims to identify the features that have a great impact on crash report reproducibility prediction. It reveals which features play key roles in the decision-making process of the model.
- RQ4: Does the multi-feature fusion phase affect the performance of PreCrash?
This research question focuses on the impact of the multi-feature fusion strategy on model performance. The goal is to verify whether fusing features from different dimensions can improve the effectiveness of the model to predict the reproducibility of crash reports.

5. Experimental Results

5.1. RQ1: How Does PreCrash Perform in the Prediction of the Reproducibility of Crash Reports?

This research question evaluates the effectiveness of PreCrash on 4133 crash reports, as described in Section 4.1. PreCrash randomly splits these 4133 crash reports into a training set, a validation set, and a testing set. Specifically, 80% of 4133 crash reports (3306) are used as the training set; 10% of 4133 crash reports (413) are used as the validation set; and 10% of 4133 crash reports (414) are used as the testing set. To evaluate the effectiveness of PreCrash in predicting crash report reproducibility, this research question compares PreCrash with five machine learning models and four deep learning models. Table 3 shows the performance of each model in terms of Accuracy, Precision, Recall, and F1-score. The results in Table 3 are obtained from a single experimental run under the report-level random split.

Table 3. Comparison of different models on the reproducibility of crash report prediction. The best result for each evaluation metric is shown in bold.

Metric	LR	RF	SVM	DT	XGBoost	Word2Vec	FastText	GloVe	BERT	PreCrash
Accuracy	0.86	0.86	0.86	0.85	0.86	0.71	0.67	0.71	0.81	0.91
Precision	0.85	0.89	0.87	0.84	0.85	0.57	0.51	0.72	0.82	0.92
Recall	0.81	0.80	0.79	0.82	0.82	0.56	0.57	0.69	0.74	0.90
F1-score	0.83	0.84	0.83	0.83	0.84	0.57	0.54	0.70	0.78	0.91

As shown in Table 3, traditional machine learning models show relatively stable performance. The Accuracy values of Logistic Regression, Random Forest, SVM, Decision Tree, and XGBoost range from 0.85 to 0.86, while their F1-scores remain between 0.83 and 0.84. These models can effectively use structured features and capture some patterns related to crash reports. However, their improvement is limited when dealing with complex semantics and long textual descriptions. In particular, Random Forest and XGBoost achieve a relatively balanced performance between Precision and Recall through ensemble learning strategies. This indicates that they have certain advantages in feature utilization and robustness. However, these methods still have limitations because they cannot fully model the implicit contextual information in crash report text, making it difficult to mine deep semantic features.

In contrast, shallow word embedding methods show relatively weak prediction performance. Word2Vec and FastText achieve Accuracy values of 0.71 and 0.67, respectively, while their F1-scores are only 0.57 and 0.54. These results indicate that they have difficulty distinguishing reproducible crash reports from non-reproducible ones. This is mainly because shallow embedding methods are limited in modeling long-distance dependencies and complex semantic relationships. Although GloVe performs better than Word2Vec and FastText, increasing the F1-score to 0.70, its overall performance is still clearly lower than that of traditional machine learning methods. The pretrained BERT model performs substantially better than shallow embedding methods. The pretrained BERT model achieves an Accuracy of 0.81 and an F1-score of 0.78, which confirms the advantage of deep bidirectional Transformers in contextual semantic modeling. However, BERT has strong semantic representation capability, which does not show an absolute advantage over traditional machine learning models. This suggests that data scale and domain-specific characteristics may affect its performance.

To assess the practical applicability of PreCrash, we conduct a preliminary efficiency evaluation. We randomly selected 100 reports from the 414 testing reports and repeated the experiment three times. PreCrash required an average of 7.25 min to complete training and prediction for 100 reports, corresponding to about 4.35 s per report. For practical reference,

we also measure the average time from issue creation to closure in the testing set under the same conditions, the average time is 11.89 days per issue. Since manually reproducing a large number of crash reports would require substantial time and effort, we do not conduct a direct human reproduction-time comparison. These results indicate that PreCrash can provide reproducibility predictions efficiently and support early crash report triage.

Overall, PreCrash achieves the best performance across all metrics. PreCrash obtains an Accuracy of 0.91, a Precision of 0.92, a Recall of 0.90, and an F1-score of 0.91. PreCrash benefits from the synergy between multi-feature fusion and the deep neural network structure. PreCrash not only uses contextual semantic features from text, but it also integrates multi-dimensional information such as environmental configurations and interaction behaviors. As a result, it achieves a substantial improvement in the crash report reproducibility prediction task. The experimental results show that PreCrash achieves the best results in terms of Accuracy, Precision, Recall, and F1-score. This verifies the effectiveness of PreCrash in the task of crash report reproducibility prediction.

5.2. RQ2: How Effective Are the Four Text Representation Models in Capturing Semantic Information from Crash Reports?

This research question aims to evaluate the effectiveness of different text representation methods in capturing semantic similarity between crash reports. Since crash reports usually contain both natural language descriptions and technical details, an effective representation can identify reports describing semantically related crash information. In RQ2, this experiment is a semantic retrieval task rather than a binary classification task. In the semantic retrieval task, each crash report is treated as a query, and five semantically related crash reports are manually selected as relevant reports for that query. The query report is compared with the candidate reports using cosine similarity, and the candidate reports are ranked in descending order according to their similarity scores. Recall@ k measures the proportion of relevant reports retrieved among the top- k results, while MRR evaluates how early the first relevant report appears in the ranking and MAP measures the overall ranking quality of all relevant reports. This retrieval experiment is used specifically to evaluate the semantic representation capability of different methods and is independent of the binary reproducibility classification evaluation. Thus, we compare four text representation methods, namely, Word2Vec, GloVe, FastText, and BERT, with the fused representation used in PreCrash.

Figure 4 shows the comparison results of four text representation models in the semantic crash report retrieval task. In the Recall@ k results shown in the left subfigure, Word2Vec, GloVe, FastText, and BERT exhibit relatively similar performance. Their Recall@5 to Recall@20 values all remain above 0.85, indicating that these methods can effectively capture semantic information from crash reports. However, there are still some differences in detail. Word2Vec achieves the highest Recall@5 value of 0.8654, suggesting that it has stronger recall capability among the top-ranked retrieval results. FastText and BERT show slight advantages in Recall@15 and Recall@20, indicating that they can capture more relevant information when the retrieval range is expanded. Overall, the differences among the four baseline models reflect the relative stability of shallow word embeddings and pretrained models in this task. It is worth noting that PreCrash substantially outperforms all baseline models from Recall@5 to Recall@20. In particular, its Recall@20 reaches 0.9841, which is much higher than those of the other models.

In terms of MRR and MAP shown in the right subfigure, the differences among the models are more pronounced. The MRR values of Word2Vec and FastText are close to 0.80, indicating relatively strong ranking capability. However, their MAP values are only around 0.56, suggesting that although they can retrieve relevant results relatively early, their overall ranking quality is still limited. GloVe performs relatively weakly, with MRR and MAP

values of 0.7745 and 0.5546, respectively. This indicates that its global statistical features do not show a clear advantage in this task. BERT achieves an MRR of 0.7857 and a MAP of 0.5692. Compared with shallow embedding methods, BERT does not show a substantial advantage. This may be because crash report texts are usually short and contain dense technical terms, which limits the full utilization of BERT's contextual modeling capability. In contrast, PreCrash achieves an MRR of 0.8711 and a MAP of 0.7591, outperforming all baseline models. This result indicates that PreCrash not only captures relevant results more quickly, but it also achieves stronger overall ranking quality and semantic representation capability. The experimental results show that the multi-feature fusion strategy of PreCrash achieves the best performance in terms of Recall@ k , MRR, and MAP. In particular, PreCrash shows substantial advantages in Recall@20 and MAP. Overall, PreCrash achieves the best performance across Recall@ k , MRR, and MAP in this semantic retrieval experiment. The results suggest that combining complementary text representations improves the semantic representation and ranking quality of crash reports.

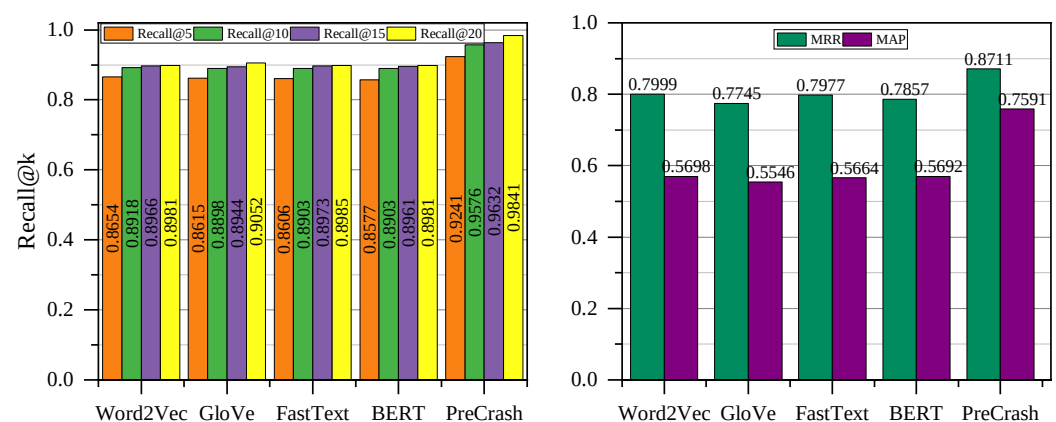


Figure 4. Comparison of different text representation models in the semantic crash report retrieval.

5.3. RQ3: Which Features Play Important Roles in the Proposed PreCrash?

This research question aims to understand the information that the model truly relies on. Feature contributions can optimize feature engineering and data collection processes, and they provide empirical guidance for developers to improve the writing quality of crash reports. This section adopts a model-based feature importance measurement method. Specifically, we evaluate the sensitivity of the trained model to individual input components through field masking and convolutional channel removal. During the field masking process, the parameters of the trained model remain unchanged; only the corresponding features are masked during evaluation, and the model is not retrained after masking. Thus, the degradation in performance resulting from this process reflects how dependent the trained model is on that feature under the original configuration. The calculation formula is as follows:

$$\Delta \text{Macro-F1}(f) = \text{Macro-F1}_{\text{baseline}} - \text{Macro-F1}_{\text{masked}(f)} \quad (31)$$

where $\text{Macro-F1}_{\text{baseline}}$ denotes the performance for all features, and $\text{Macro-F1}_{\text{masked}(f)}$ denotes the performance after feature f is masked or ablated. A larger value of $\Delta \text{Macro-F1}(f)$ indicates that the model performance is sensitive to the removal of feature f , suggesting a strong dependence of the model on this input.

In addition, Macro-F1 is defined as the average F1-score over all classes. It is used to measure the overall performance of the model under class imbalance. The calculation formula is as follows:

$$\text{Macro-F1} = \frac{1}{C} \sum_{i=1}^C \frac{2 \cdot P_i \cdot R_i}{P_i + R_i} \quad (32)$$

where C denotes the number of classes, and P_i and R_i denote the Precision and Recall of the i -th class, respectively. Based on this analysis, this study not only improves the interpretability and transparency of the model, but it also helps identify the report fields on which the model shows relatively strong predictive dependence. This can guide the community to optimize information quality, shorten the repair cycle, and enhance the reliability of software systems.

Figure 5 presents the feature importance analysis of different features in model prediction. The results show that masking Feature #48 (Steps to reproduce) produces the largest decrease in Macro-F1 among the evaluated features, with a reduction of 0.0915. This result suggests that the current PreCrash model shows relatively strong dependence on the information provided by reproduction steps. This indicates that clear reproduction steps not only help human analysts understand crash issues, but they are also crucial for the model to capture effective patterns. Temporary fixes and crash severity also show high importance, with Macro-F1 decreases of 0.0532 and 0.0475, respectively. The observed performance decreases suggest that the current model also uses information related to temporary fixes and crash severity during prediction.

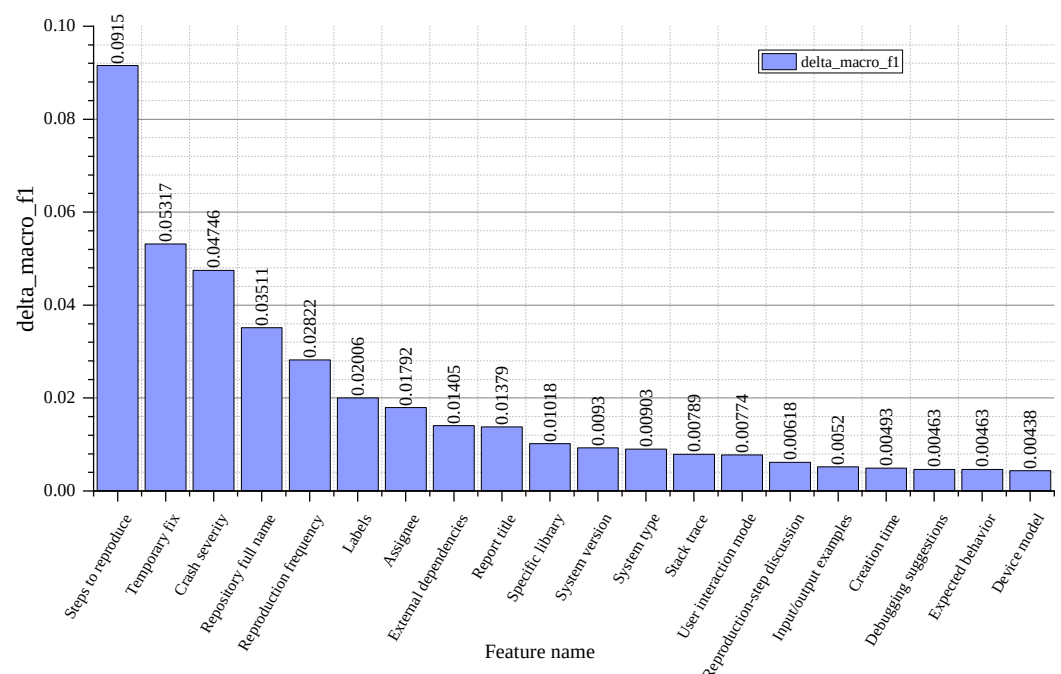


Figure 5. Feature importance analysis of different features in the proposed PreCrash.

In addition, features such as repository full name, reproduction frequency, and labels also show moderate importance. This indicates that metadata can help the model better distinguish reproducible crash reports from non-reproducible ones to some extent. Meanwhile, some environmental or basic metadata features, such as creation time, system type, and device model, contribute relatively little to performance improvement. Their Macro-F1 changes after ablation are close to 0. The low contribution of these features indicates that general environmental descriptions alone are insufficient to provide strong discriminative evidence for the model. Overall, the model relies more on features that directly describe issue details and developer feedback, while it depends less on general environmental parameters. Overall, the masking results indicate that PreCrash shows greater predictive dependence on reproduction steps and developer-feedback-related information than on several general metadata features. These observations describe the behavior of the trained model and should not be interpreted as evidence that these features causally determine crash reproducibility.

5.4. RQ4: Does the Multi-Feature Fusion Phase Affect the Performance of PreCrash?

The multi-feature fusion stage can capture rich semantic features from crash reports. This research question aims to evaluate the impact of multi-feature fusion on the PreCrash approach. To this end, this section removes the four text representations in PreCrash one by one. They are denoted as *PreCrash_Word2Vec*, *PreCrash_GloVe*, *PreCrash_FastText*, and *PreCrash_BERT*, respectively. Then, all four text representations are removed and the corresponding variant is denoted as *PreCrash_MFF*.

Table 4 presents the performance changes of the model after different text representation methods are removed. Specifically, after removing the Word2Vec representation, the model accuracy decreases to 0.81, while the Precision and Recall are 0.88 and 0.82, respectively, and the F1-score is 0.85. After removing the GloVe representation, the accuracy is 0.83, while the Precision and Recall are 0.85 and 0.88, respectively, and the F1-score is 0.87. After removing the FastText representation, the accuracy reaches 0.89, while the Precision and Recall are 0.83 and 0.89, respectively, and the F1-score is 0.86. After removing the BERT representation, the accuracy is 0.87, while the Precision and Recall are 0.81 and 0.83, respectively, and the F1-score is 0.82. Finally, after removing the multi-feature fusion framework (MFF), the model performance decreases substantially. The accuracy drops to only 0.72, while the Precision and Recall are 0.70 and 0.69, respectively, and the F1-score decreases to 0.70.

Table 4. Ablation results of different text representations

Metric	<i>PreCrash_Word2Vec</i>	<i>PreCrash_GloVe</i>	<i>PreCrash_FastText</i>	<i>PreCrash_BERT</i>	<i>PreCrash_MFF</i>
Accuracy	0.81	0.83	0.89	0.87	0.72
Precision	0.88	0.85	0.83	0.81	0.70
Recall	0.82	0.88	0.89	0.83	0.69
F1-score	0.85	0.87	0.86	0.82	0.70

The experimental results in Table 4 show that different text representation methods have different effects within the multi-feature fusion framework. Overall, Word2Vec and GloVe show relatively stable performance in the fusion model. FastText achieves an accuracy of 0.89 and maintains relatively high Recall and F1-score values. These results indicate that it has good adaptability in capturing semantic features from crash reports. In comparison, GloVe achieves a more balanced performance between Precision and Recall, reflecting its advantage in general semantic representation. FastText also shows relatively strong overall performance. Its accuracy and Recall are close to those of Word2Vec, but its Precision and F1-score are slightly lower. This suggests that subword-level feature modeling improves robustness to word-form variations to some extent, while this advantage is not fully converted into prediction performance.

To evaluate the generalizability of PreCrash, we conduct ablation experiments by removing selected features and project subsets. Table 5 shows ablation results for potentially label-related features and project types. In ablation experiments of w/o Feature#48, w/o Feature (#32, #34, #38), and w/o Feature (#48, #32, #34, #38), the experimental setup is kept the same as that of the original PreCrash model. For the Android-only project evaluation, all Java projects are excluded and the remaining 42 Android projects are divided into approximately 34 training projects, 4 validation projects, and 4 testing projects. For the Java-only project evaluation, all Android projects are excluded and the remaining 60 Java projects are divided into 48 training projects, 6 validation projects, and 6 testing projects. Each experiment is repeated five times, and the final results are reported as the mean $\pm$ standard deviation.

Table 5. Ablation results for potentially label-related features and project types.

Model Setting	Accuracy	Precision	Recall	F1-Score
w/o Feature#48	0.91 ± 0.004	0.91 ± 0.005	0.90 ± 0.006	0.91 ± 0.003
w/o Feature (#32, #34, #38)	0.89 ± 0.005	0.87 ± 0.003	0.88 ± 0.004	0.87 ± 0.004
w/o Feature (#48, #32, #34, #38)	0.89 ± 0.005	0.87 ± 0.004	0.87 ± 0.005	0.87 ± 0.003
Android-only projects	0.78 ± 0.006	0.79 ± 0.005	0.76 ± 0.005	0.77 ± 0.003
Java-only projects	0.69 ± 0.008	0.70 ± 0.007	0.68 ± 0.007	0.69 ± 0.006
PreCrash	0.91 ± 0.003	0.92 ± 0.005	0.90 ± 0.004	0.91 ± 0.003

In Table 5, the full PreCrash model achieves an Accuracy of 0.91 ± 0.003 and an F1-score of 0.91 ± 0.003 . Removing Feature #48 (Steps to reproduce) causes almost no performance degradation, with Accuracy and F1-score remaining at 0.91 ± 0.004 and 0.91 ± 0.003 , respectively. It is worth noting that this experimental setup differs from that in RQ3. In RQ3, masking Feature#48 from the trained model resulted in the large drop in the Macro-F1 score, indicating that the full model heavily relies on information from the reproduction step. However, in Table 5, the model is retrained after removing Feature#48, allowing the remaining features to compensate for its absence. Thus, Feature#48 is important for the trained, complete model; however, it is not indispensable after retraining. After removing the three potentially outcome-related comment features (#32, #34, and #38), the Accuracy decreases to 0.89 ± 0.005 and the F1-score to 0.87 ± 0.004 . Removing Feature #48 together with these three comment-related features yields similar results, with an Accuracy of 0.89 ± 0.005 and an F1-score of 0.87 ± 0.003 . These results indicate that the identified features provide useful predictive information, while the performance of PreCrash is not primarily driven by them. We further evaluate the effect of project composition. Removing Java projects reduces the Accuracy and F1-score to 0.78 ± 0.006 and 0.77 ± 0.003 , respectively, whereas removing Android projects results in a larger decrease, to 0.69 ± 0.008 and 0.69 ± 0.006 .

Although BERT generally performs well in many natural language processing tasks, its Accuracy and F1-score in this experiment are lower than those achieved by traditional word embedding methods. This may be related to the domain-specific characteristics of crash reports. The pretrained model may fail to fully capture task-related domain knowledge. In addition, *PreCrash_{MFF}* performs worse than the variants that combine a single text representation with structured features across all metrics. This result indicates that simply removing the fusion mechanism causes a substantial loss of complementary information. It also suggests that effective feature selection and fusion mechanisms are more important than directly stacking all available feature sources without considering their contribution. The experimental results show that traditional embeddings such as Word2Vec and FastText perform relatively well in this task, while the removal of the multi-feature fusion framework leads to a substantial performance degradation. This indicates that, when designing prediction models, researchers should pay more attention to the rationality of feature selection and fusion mechanisms rather than relying only on the accumulation of feature quantity. Overall, the results show that PreCrash may benefit from complementary information across multiple report features and heterogeneous project types, rather than relying on a small set of potentially label-related features.

5.5. Discussion

Experimental results show that PreCrash benefits from complementary information across multiple crash report features and text representations. To further understand the failure cases of PreCrash, we manually examined representative misclassified reports in the testing set. For example, Issue#2987 of Project Google_guava project is a false-positive case. The report contains relatively detailed crash descriptions and logs and is predicted

by PreCrash as reproducible. However, it lacks sufficient environmental and dependency information and is manually labeled as non-reproducible. In contrast, Issue#35529 of Project Keycloak_keycloak project is a false-negative case. The report contains limited reproduction steps and sparse environmental information and is predicted by PreCrash as non-reproducible. However, additional evidence in the report supports its reproducibility. These cases suggest that incomplete information and ambiguous environmental conditions are associated with several prediction errors. Overall, the results suggest that PreCrash benefits from complementary information across multiple report dimensions, while more effective modeling of missing and uncertain information may further improve its performance.

In this study, we adopt a direct feature fusion strategy rather than introducing attention mechanisms. The main purpose of PreCrash is to investigate crash report reproducibility prediction and evaluate whether complementary information from multiple feature representations can improve prediction performance. A simple fusion strategy reduces additional model complexity and makes the contribution of different representations easier to analyze through ablation experiments. Although attention-based fusion mechanisms may dynamically assign different weights to heterogeneous information, they also introduce additional parameters and training complexity. Thus, we leave adaptive fusion mechanisms, such as attention and gating, as an important direction for future work.

We further summarize several common missing patterns in non-reproducible crash reports. These patterns mainly include missing or ambiguous reproduction steps, unspecified software or dependency versions, incomplete runtime environment information, missing input data or configuration parameters, and insufficient descriptions of the conditions that trigger the crash. Such missing information makes it difficult for developers to reconstruct the original execution environment and reproduce the reported failure. Incomplete crash reports may introduce selection bias. This study focuses on reports with sufficient information for feature extraction, while handling highly incomplete reports remains an important direction for future work.

6. Threats to Validity

This section discusses the main threats to the validity of PreCrash from three perspectives: construct validity, internal validity, and external validity.

Construct validity. This threat concerns the reference labels used for crash report reproducibility. In this study, the labels are manually assessed based on the information available in crash reports rather than obtained by actually reproducing every crash in its original execution environment. Although two authors independently assess each report and a third author resolves disagreements, the resulting labels represent human assessments of reproducibility based on report evidence rather than objective reproduction outcomes. Therefore, a report assessed as reproducible does not necessarily guarantee that the corresponding crash can be reproduced in practice under every execution environment.

Another potential threat is label leakage from reproduction-related discussions and other information accumulated during the lifecycle of an issue. Comments, version-related information, labels, or other fields available after the initial report may contain information correlated with the final reproducibility assessment. To reduce this risk, reproduction-related features are defined to capture contextual information without explicitly encoding whether a reproduction attempt succeeds or fails. We also conduct additional retraining-based ablation experiments by removing potentially label-related Features #48, #32, #34, and #38. When all four features are removed, PreCrash obtains an F1-score of 0.87 ± 0.003 , compared with 0.91 ± 0.003 for the full model. Nevertheless, indirect outcome-related information may still be present in natural language discussions or other issue fields, and this remains a limitation of the current study.

Internal validity. The main experiment uses a report-level random split. Therefore, reports from the same GitHub project may appear in both the training and testing subsets. Because reports from the same project may share templates, terminology, labels, version formats, and other project-specific characteristics, such overlap may produce an optimistic estimate of performance on previously unseen projects. To examine this threat, we additionally conduct a project-level evaluation in which all reports from the same project are assigned to the same subset, preventing project overlap among the training, validation, and testing sets.

To reduce the dependence of the additional robustness analyses on a single training run, the retraining-based feature-ablation and project-level experiments reported in Table 5 are repeated five times using different random seeds, and their results are reported as the mean $\pm$ standard deviation. The main comparison reported in Table 3, corresponds to the original single-run evaluation under the report-level random split. Thus, some variation associated with stochastic model training may remain in the main comparison. The feature-masking and ablation experiments allow us to observe how model performance changes when individual input components are removed. However, these results characterize the dependence of the trained model on particular inputs and should not be interpreted as evidence of causal relationships between individual features and crash reproducibility. Moreover, correlations among input features may affect the observed importance of individual features.

External validity. The data-cleaning procedure may introduce selection bias. Reports with 25 or more missing features are removed because more than half of the predefined feature information is unavailable. However, highly incomplete reports may represent particularly difficult reproducibility cases. Thus, the current evaluation mainly reflects crash reports with sufficient information for feature extraction, and the applicability of PreCrash to highly incomplete reports requires further investigation.

Our dataset is collected from 102 open-source Java and Android projects on GitHub. Although these projects cover two major software ecosystems, the results may not directly generalize to projects written in other programming languages, closed-source software, industrial issue-tracking systems, or other development environments. Future studies should evaluate PreCrash on broader software ecosystems and independent industrial datasets.

7. Related Work

We present the related work in two categories, crash report analysis and crash reproduction.

Crash report analysis. Stack traces are important clues for fault localization [36,37]. In practical applications, after developers obtain a crash report, they first extract the stack trace information and normalize it to reduce the impact of environmental differences. Then, they analyze the stack trace to generate a crash, which is used to uniquely identify this type of crash. For example, Dang et al. [38] proposed ReBucket, an automatic crash report bucketing approach based on call stack similarity. Their method outperforms the heuristic methods available at that time. Vasiliev et al. [39] proposed TraceSim, which combines TF-IDF, Levenshtein distance, and machine learning to measure stack trace similarity to improve the quality of crash report analysis. Crash reports are clustered and deduplicated so that identical or similar crashes can be grouped together. Then, these reports are prioritized according to factors such as impact scope and occurrence frequency, which helps developers allocate maintenance resources effectively. For example, Rodrigues et al. [40] proposed FaST, a linear-time stack alignment heuristic for accelerating similarity computation in crash bucketing. Their results show that FaST can improve the throughput of online systems while maintaining near-optimal alignment quality. Finally,

these analysis results are applied to fault localization and help developers quickly locate the root causes of defects. Duplicate crash reports undoubtedly hinder the analysis process of crash reports. For example, Wu et al. [41] expanded crash stacks using functions in static call graphs to infer possible crash traces, and their results show that this method can effectively improve function-level localization without requiring test coverage information. Gu et al. [42] used a binary classification model to determine whether a fault is located within the stack-reachable scope, and their results show that this method can accelerate the selection of subsequent localization strategies. Liu et al. [7] explored nested exceptions in Java projects and analyzed root causes of nested exceptions.

Crash reproduction. In crash reproduction, developers first extract the crash reproduction steps described in natural language from crash reports. Then, they perform text parsing and entity extraction to identify key actions and components, such approaches include Yakusu [43] and ReCDroid [44]. Next, these entities are organized into step sequences, and missing-step detection is performed to identify key operations that may be omitted from the description. Zhang et al. [45] used reinforcement learning and natural language processing techniques to detect missing steps in crash reproduction descriptions to achieve efficient matching between crash steps and UI components. Then, the complete step sequences are transformed into executable scripts for automated crash reproduction and validation. Wendland et al. [46] proposed AndroR2, a dataset containing 90 manually validated mobile bug reports, reproducible APKs, scripts, and videos. This dataset supports objective evaluation of the process from natural language crash reproduction steps to crash reproduction. In addition, Moran et al. [47] proposed CrashScope, an automated reproduction approach based on screenshots and videos in crash reports. CrashScope can automatically generate reproducible scripts to improve the efficiency of manual reproduction. These studies may help developers validate and reproduce crash scenarios reported by users.

8. Conclusions

In this paper, we propose PreCrash, a multi-feature fusion approach for predicting the reproducibility of crash reports. PreCrash treats the reproducibility prediction of crash reports as a binary classification task. PreCrash extracts 49 features from each crash report, including basic metadata, contextual and environmental information, comment and interaction features, and supplementary evidence. To capture textual semantics, PreCrash combines Word2Vec, GloVe, FastText, and BERT representations. The four text representations are fused into a unified representation, which is then fed into a TextCNN-based prediction model. We evaluate PreCrash on a dataset containing 4133 crash reports collected from 102 real-world GitHub projects. The experimental results show that PreCrash achieves the best performance among compared methods, with an accuracy of 0.91 and an F1-score of 0.91.

In future work, we plan to extend the dataset to include various programming languages, software ecosystems, and industrial issue-tracking platforms to further evaluate the generalizability of our approach. We also plan to investigate an interpretable prediction mechanism to provide developers with explicit explanations. These extensions can further enhance the practical value of PreCrash in software maintenance and automated debugging.

Author Contributions: J.X. contributed to the conceptualization. S.L. and Z.S. contributed to the methodology of this study. S.L. developed the software. S.L. and H.X. contributed to the validation and investigation. S.L. wrote the original draft. Z.S. and J.X. reviewed and edited the draft. S.L. conducted the visualization. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Natural Science Foundation of China, Grant Nos. 62572363 and 62502352.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study and the prototype of PreCrash are publicly available at <https://github.com/lusaty/PreCrash> (accessed on 8 September 2026).

Acknowledgments: During the preparation of this manuscript, the authors used ChatGPT (OpenAI, GPT-5) solely for English editing and readability improvement. We have not used large language models or other AI tools in designing and analyzing methods or experiments. After using this tool, the authors reviewed and revised the content. The authors take full responsibility for the content of the manuscript.

Conflicts of Interest: The authors declare no conflicts of interest.

References

- Huang, Y.; Wang, J.; Liu, Z.; Li, M.; Wang, S.; Chen, C.; Hu, Y.; Wang, Q. One Sentence Can Kill the Bug: Auto-Replay Mobile App Crashes From One-Sentence Overviews. *IEEE Trans. Softw. Eng.* **2025**, *51*, 975–989. [\[CrossRef\]](#)
- Su, T.; Fan, L.; Chen, S.; Liu, Y.; Xu, L.; Pu, G.; Su, Z. Why My App Crashes? Understanding and Benchmarking Framework-Specific Exceptions of Android Apps. *IEEE Trans. Softw. Eng.* **2022**, *48*, 1115–1137. [\[CrossRef\]](#)
- Wang, Y.; Meijer, W.; López, J.A.H.; Nilsson, U.; Varró, D. Why Do Machine Learning Notebooks Crash? An Empirical Study on Public Python Jupyter Notebooks. *IEEE Trans. Softw. Eng.* **2025**, *51*, 2181–2196. [\[CrossRef\]](#)
- Liu, Z.; Li, C.; Chen, C.; Wang, J.; Chen, M.; Wu, B.; Wang, Y.; Hu, J.; Wang, Q. Seeing is Believing: Vision-Driven Non-Crash Functional Bug Detection for Mobile Apps. *IEEE Trans. Softw. Eng.* **2025**, *51*, 3452–3466. [\[CrossRef\]](#)
- Xu, Y.; Liu, C.; Li, Y.; Xie, Q.; Choi, H. A Method of Component Prediction for Crash Bug Reports Using Component-Based Features and Machine Learning. In *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2023, Taipa, Macao, 22–24 March 2023*; Zhang, T., Xia, X., Novielli, N., Eds.; IEEE: Piscataway, NJ, USA, 2023; pp. 773–777. [\[CrossRef\]](#)
- Zhao, Y.; Su, T.; Liu, Y.; Zheng, W.; Wu, X.; Kavuluru, R.; Halfond, W.G.J.; Yu, T. ReCDroid+: Automated End-to-End Crash Reproduction from Bug Reports for Android Apps. *ACM Trans. Softw. Eng. Methodol.* **2022**, *31*, 36:1–36:33. [\[CrossRef\]](#)
- Liu, S.; Xu, H.; Xin, Q.; Xuan, J. Exceptions Can Be Nested: An Exploratory Study on Nested Exceptions in Java Crashes. *J. Softw. Evol. Process* **2026**, *38*, e70099. [\[CrossRef\]](#)
- Remil, Y.; Bendimerad, A.; Mathonat, R.; Raïssi, C.; Kaytoue, M. DeepLSH: Deep Locality-Sensitive Hash Learning for Fast and Efficient Near-Duplicate Crash Report Detection. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, 14–20 April 2024*; ACM: New York, NY, USA, 2024; pp. 198:1–198:12. [\[CrossRef\]](#)
- Runeson, P.; Alexandersson, M.; Nyholm, O. Detection of Duplicate Defect Reports Using Natural Language Processing. In *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, 20–26 May 2007*; IEEE Computer Society: Los Alamitos, CA, USA, 2007; pp. 499–510. [\[CrossRef\]](#)
- Wang, B.; Deng, M.; Chen, M.; Lin, Y.; Zhou, J.; Zhang, J.M. Assessing the effectiveness of recent closed-source large language models in fault localization and automated program repair. *Autom. Softw. Eng.* **2026**, *33*, 26. [\[CrossRef\]](#)
- Deng, Y.; Misir, M.; Chen, J.; Xuan, J. SustainSR: Interpretable OSS Sustainability Prediction via Symbolic Regression. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE 2026), Munich, Germany, 12–16 October 2026*; ACM: New York, NY, USA, 2026.
- Medeiros, M.; Kulesza, U.; Bonifácio, R.; Barbosa, E.A.; Coelho, R. Improving Bug Localization by Mining Crash Reports: An Industrial Study. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution, ICSME 2020, Adelaide, Australia, 28 September–2 October 2020*; IEEE: Piscataway, NJ, USA, 2020; pp. 766–775. [\[CrossRef\]](#)
- Wu, R.; Wen, M.; Cheung, S.; Zhang, H. ChangeLocator: Locate crash-inducing changes based on crash reports. *Empir. Softw. Eng.* **2018**, *23*, 2866–2900. [\[CrossRef\]](#)
- Rahman, M.T.; Rigby, P.C. The impact of failing, flaky, and high failure tests on the number of crash reports associated with Firefox builds. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, 4–9 November 2018*; Leavens, G.T., Garcia, A., Pasareanu, C.S., Eds.; ACM: New York, NY, USA, 2018; pp. 857–862. [\[CrossRef\]](#)

15. Maiga, A.; Hamou-Lhadj, A.; Nayrolles, M.; Sabor, K.K.; Larsson, A. An empirical study on the handling of crash reports in a large software company: An experience report. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, 29 September–1 October 2015*; Koschke, R., Krinke, J., Robillard, M.P., Eds.; IEEE Computer Society: Piscataway, NJ, USA, 2015; pp. 342–351. [\[CrossRef\]](#)
16. Ahmed, I.; Mohan, N.; Jensen, C. The Impact of Automatic Crash Reports on Bug Triaging and Development in Mozilla. In *Proceedings of the International Symposium on Open Collaboration, OpenSym 2014, Berlin, Germany, 27–29 August 2014*; Riehle, D., González-Barahona, J.M., Robles, G., Möslin, K.M., Schieferdecker, I., Cress, U., Wichmann, A., Hecht, B.J., Jullien, N., Eds.; ACM: New York, NY, USA, 2014; pp. 1:1–1:8. [\[CrossRef\]](#)
17. Rahman, M.M.; Khomh, F.; Castelluccio, M. Works for Me! Cannot Reproduce—A Large Scale Empirical Study of Non-reproducible Bugs. *Empir. Softw. Eng.* **2022**, *27*, 111. [\[CrossRef\]](#)
18. Curto, G.; Acosta, M.F.J.; Comim, F.; Zapirain, B.G. Are AI systems biased against the poor? A machine learning analysis using Word2Vec and GloVe embeddings. *AI Soc.* **2024**, *39*, 617–632. [\[CrossRef\]](#) [\[PubMed\]](#)
19. Vo, T.; Le, Q.P.; Jung, H.; Choi, J.; Vu, T.T.H.; Doan, V.H.M.; Mondal, S.; Oh, J. Multisensor Smart Glove With Unsupervised Learning Model for Real-Time Wrist Motion Analysis in Golf Swing Biomechanics. *IEEE Internet Things J.* **2025**, *12*, 16574–16586. [\[CrossRef\]](#)
20. Alvi, M.A.; Jalil, Z.; Imran, M.; Ayub, M. RGV2: A Framework for crypto-ransomware detection using FastText and neural networks. *Knowl. Inf. Syst.* **2026**, *68*, 21. [\[CrossRef\]](#)
21. de Souza, F.C.; Nogueira, R.F.; Lotufo, R.A. BERT models for Brazilian Portuguese: Pretraining, evaluation and tokenization analysis. *Appl. Soft Comput.* **2024**, *149*, 110901. [\[CrossRef\]](#)
22. Fan, Y.; Xia, X.; Lo, D.; Hassan, A.E. Chaff from the Wheat: Characterizing and Determining Valid Bug Reports. *IEEE Trans. Softw. Eng.* **2020**, *46*, 495–525. [\[CrossRef\]](#)
23. Zimmermann, T.; Premraj, R.; Bettenburg, N.; Just, S.; Schröter, A.; Weiss, C. What Makes a Good Bug Report? *IEEE Trans. Softw. Eng.* **2010**, *36*, 618–643. [\[CrossRef\]](#)
24. Kim, D.; Wang, X.; Kim, S.; Zeller, A.; Cheung, S.; Park, S. Which Crashes Should I Fix First?: Predicting Top Crashes at an Early Stage to Prioritize Debugging Efforts. *IEEE Trans. Softw. Eng.* **2011**, *37*, 430–447. [\[CrossRef\]](#)
25. Soltani, M.; Derakhshanfar, P.; Devroey, X.; van Deursen, A. A benchmark-based evaluation of search-based crash reproduction. *Empir. Softw. Eng.* **2020**, *25*, 96–138. [\[CrossRef\]](#)
26. Chen, N.; Kim, S. STAR: Stack Trace Based Automatic Crash Reproduction via Symbolic Execution. *IEEE Trans. Softw. Eng.* **2015**, *41*, 198–220. [\[CrossRef\]](#)
27. McCully, G.A.; Hastings, J.D.; Xu, S.; Fortier, A. Comparing Unidirectional, Bidirectional, and Word2vec Models for Discovering Vulnerabilities in Compiled Lifted Code. In *Proceedings of the 13th International Symposium on Digital Forensics and Security, ISDFS 2025, Boston, MA, USA, 24–25 April 2025*; IEEE: Piscataway, NJ, USA, 2025; pp. 1–6. [\[CrossRef\]](#)
28. Nagaraju, K.C.; Reddy, C.R.K. Reusable Component Retrieval from a Large Repository Using Word2Vec with Continuous Bag of Words. *Ing. Syst. Inf.* **2021**, *26*, 453–460. [\[CrossRef\]](#)
29. Sruthi, K.; Naik, S.M. A novel framework for phishing attack detection using domain-adapted GloVe embeddings and attention-enhanced neural sequence model. *Appl. Soft Comput.* **2026**, *188*, 114441. [\[CrossRef\]](#)
30. Wang, C.; Sun, L.; Wang, X.; Bai, X. Intent-Based Medical Transport Classification Using Transformer Networks for Intelligent Transportation Systems. *IEEE Internet Things J.* **2026**, *13*, 18213–18225. [\[CrossRef\]](#)
31. Ye, L.; Wen, M.; Zhang, F.; Xu, Y.; Zhang, Y.; Ma, Y.; Wen, K. Hybrid Quantum-Classical Algorithm for an Integrated Feature Selection and Logistic Regression Model. *IEEE Trans. Eng. Manag.* **2026**, *73*, 1807–1819. [\[CrossRef\]](#)
32. Bounia, L.; Setitra, I. Enhancing the intelligibility of decision trees with concise and reliable probabilistic explanations. *Data Knowl. Eng.* **2025**, *156*, 102394. [\[CrossRef\]](#)
33. Nwafor, C.N.; Nwafor, O.; Brahma, S.; Acharyya, M. A hybrid FAIR and XGBoost framework for cyber-risk intelligence and expected loss prediction. *Expert Syst. Appl.* **2026**, *299*, 129920. [\[CrossRef\]](#)
34. Sujon, K.M.; Hassan, R.; Choi, K.; Samad, M.A. Accuracy, precision, recall, f1-score, or MCC? empirical evidence from advanced statistics, ML, and XAI for evaluating business predictive models. *J. Big Data* **2025**, *12*, 268. [\[CrossRef\]](#)
35. Song, J.; Xiao, J.; Wu, F.; Wu, H.; Zhang, T.; Zhang, Z.M.; Zhu, W. Hierarchical contextual attention recurrent neural network for map query suggestion. *IEEE Trans. Knowl. Data Eng.* **2017**, *29*, 1888–1901. [\[CrossRef\]](#)
36. Kampel, M.; Sabetzadeh, M.; Knauss, E.; Köppe, T. An exploration of combinatorial testing-based approaches to fault localization for explainable AI. *Ann. Math. Artif. Intell.* **2022**, *90*, 951–964. [\[CrossRef\]](#)
37. Chen, H.; Zhao, Z.; Shuai, Z.; Xuan, J. Testing Explainability of Chain of Thought for Large Language Models. *Appl. Sci.* **2026**, *16*, 3112. [\[CrossRef\]](#)
38. Dang, Y.; Wu, Y.; Zhang, Y.; Zheng, X.; Nobel, P. ReBucket: A method for clustering duplicate crash reports based on call stack similarity. In *Proceedings of the 34th International Conference on Software Engineering (ICSE), Zurich, Switzerland, 2–9 June 2012*; pp. 1084–1093. [\[CrossRef\]](#)

39. Vasiliev, R.; Koznov, D.V.; Chernishev, G.A.; Khvorov, A.; Luciv, D.V.; Povarov, N. TraceSim: A method for calculating stack trace similarity. In *Proceedings of the MaLTesQuE@ESEC/FSE 2020*, Virtual, 13 November 2020; pp. 25–30. [\[CrossRef\]](#)
40. Rodrigues, I.M.; Aloise, D.; Fernandes, E.R. FaST: A linear time stack trace alignment heuristic for crash report deduplication. In *Proceedings of the 19th International Conference on Mining Software Repositories (MSR)*, Pittsburgh, PA, USA, 23–24 May 2022; pp. 549–560. [\[CrossRef\]](#)
41. Wu, R.; Zhang, H.; Cheung, S.-C.; Kim, S. CrashLocator: Locating crashing faults based on crash stacks. In *Proceedings of the 23rd International Symposium on Software Testing and Analysis (ISSTA)*, San Jose, CA, USA, 21–25 July 2014; pp. 204–214. [\[CrossRef\]](#)
42. Gu, Y.; Xuan, J.; Zhang, H.; Zhang, L.; Fan, Q.; Xie, X.; Qian, T. Does the fault reside in a stack trace? Assisting crash localization by predicting crashing fault residence. *J. Syst. Softw.* **2019**, *148*, 88–104. [\[CrossRef\]](#)
43. Fazzini, M.; Prammer, M.; d’Amorim, M.; Orso, A. Automatically translating bug reports into test cases for mobile apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, 16–21 July 2018*; ACM: New York, NY, USA, 2018; pp. 141–152.
44. Zhao, Y.; Yu, T.; Su, T.; Liu, Y.; Zheng, W.; Zhang, J.; Halfond, W.G.J. ReCDroid: Automatically reproducing Android application crashes from bug reports. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, 25–31 May 2019*; IEEE/ACM: Piscataway, NJ, USA, USA, 2019; pp. 128–139.
45. Zhang, Z.; Winn, R.; Zhao, Y.; Yu, T.; Halfond, W.G.J. Automatically Reproducing Android Bug Reports using Natural Language Processing and Reinforcement Learning. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, 17–21 July 2023*; ACM: New York, NY, USA, 2023; pp. 411–422.
46. Wendland, T.; Sun, J.; Mahmud, J.; Mansur, S.M.H.; Huang, S.; Moran, K.; Rubin, J.; Fazzini, M. Andror2: A Dataset of Manually-Reproduced Bug Reports for Android apps. In *Proceedings of the 18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, 17–19 May 2021*; IEEE: Piscataway, NJ, USA, 2021; pp. 600–604.
47. Moran, K.; Vásquez, M.L.; Bernal-Cárdenas, C.; Vendome, C.; Poshyvanyk, D. CrashScope: A practical tool for automated testing of Android applications. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, 20–28 May 2017—Companion Volume*; IEEE Computer Society: Piscataway, NJ, USA, 2017; pp. 15–18.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.