

Article

Enhancing Bug Assignment with Developer-Specific Feature Extraction and Hybrid Deep Learning

Geunseok Yang ^{1,*}, Jinfeng Ji ² and Dongkyu Kim ³

¹ Department of Computer Applied Mathematics, Computer System Institute, Hankyong National University, Anseong 17579, Republic of Korea

² Department of Computer Applied Mathematics, Hankyong National University, Anseong 17579, Republic of Korea; jijinfeng@hknu.ac.kr

³ Department of Computer Engineering, Kyungnam University, Changwon 51767, Republic of Korea; dongkyukim@kyungnam.ac.kr

* Correspondence: gsyang@hknu.ac.kr

Abstract: The increasing reliance on software in diverse domains has led to a surge in user-reported functional enhancements and unexpected bugs. In large-scale open-source projects like Eclipse and Mozilla, initial bug assignment frequently faces challenges, with approximately 50% of bug reports being reassigned due to the inability of the initially assigned developer to resolve the issue effectively. This reassignment process contributes to elevated software maintenance costs and delays in bug resolution. To address this, we propose a developer recommendation model that assigns the most suitable developer for a given bug report at the outset, thereby minimizing reassignment rates. Our approach combines a top-K feature selection algorithm tailored for each developer with a hybrid Convolutional Neural Network–Long Short-Term Memory (CNN–LSTM) architecture to capture the nuanced patterns in bug reports and developer expertise. The model was evaluated on prominent open-source projects, including Google Chrome, Mozilla Core, and Mozilla Firefox. Experimental results show that the proposed model significantly outperforms baseline approaches, with an improvement in developer recommendation accuracy of approximately 0.3582 when comparing the best-performing configuration to the worst-performing configuration of our model. Furthermore, the baseline difference was reduced by approximately 0.1343. A statistical analysis confirms the significant performance improvement achieved by the proposed method over existing baselines. These findings underscore the potential of our model to enhance efficiency in bug resolution workflows, reduce maintenance costs, and improve overall software quality in open-source ecosystems.

Keywords: developer recommendation; developer-based feature selection; CNN–LSTM algorithm; software evolution; software engineering



Academic Editors: Krzysztof Szczypiorski and Ping-Feng Pai

Received: 15 May 2025

Revised: 14 June 2025

Accepted: 17 June 2025

Published: 19 June 2025

Citation: Yang, G.; Ji, J.; Kim, D. Enhancing Bug Assignment with Developer-Specific Feature Extraction and Hybrid Deep Learning.

Electronics **2025**, *14*, 2493.

<https://doi.org/10.3390/electronics14122493>

Copyright: © 2025 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) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

As software becomes increasingly embedded across a wide range of domains, the volume and complexity of user-reported bugs and feature requests continue to grow. In large-scale open-source projects, such as Google Chrome and Mozilla Firefox, bug reports are typically reviewed manually by project managers who assign developers to address them. However, approximately 40–50% of these initial assignments are later reassigned due to mismatches in expertise or availability [1,2]. This high reassignment rate causes delays in bug resolution, increases the maintenance costs, and ultimately degrades the overall software quality.

To address these challenges, efficient and accurate developer assignment is essential for streamlining software maintenance workflows and improving system reliability. Automated developer recommendation has emerged as a promising approach to reduce manual overhead and subjectivity. While prior models have achieved meaningful progress, many still face limitations in scalability, generalizability, and contextual understanding.

For example, Guo et al. [3] proposed a convolutional neural network (CNN)-based model that incorporates word embeddings and batch normalization. Jahanshahi et al. [4] introduced a dependency-aware triaging approach using natural language processing and integer programming, and Park et al. [5] combined case-based reasoning (CBR) with collaborative filtering (CF). Although these models improve accuracy, they often rely on static feature sets or single-model architectures that struggle to generalize across datasets and fully capture the complexity of bug report data.

More recently, transformer-based architectures such as BERT have achieved strong performances across a wide range of natural language processing tasks. However, applying these models to developer recommendations presents practical challenges. Transformers typically require extensive fine-tuning and high computational costs, which may limit their use in project-specific or resource-constrained settings. Furthermore, they process the input text holistically and may overlook developer-specific patterns, which are crucial for personalized bug assignment. As a result, their effectiveness in this domain may be limited.

In contrast, we propose a novel and adaptive developer recommendation method that integrates a developer-specific top-K feature selection algorithm [6] with a hybrid CNN-Long Short-Term Memory (CNN-LSTM) [7] architecture. The top-K component dynamically identifies the most relevant textual features from bug reports for each developer, reducing noise and enhancing personalization. The CNN module captures spatial dependencies, while the LSTM module models the sequential patterns in bug report content. Together, these components enable the model to learn developer-specific expertise in a computationally efficient manner that is well-suited for structured bug data.

We evaluated the proposed method on large-scale datasets [8] from open-source projects, including Google Chrome, Mozilla Core, and Mozilla Firefox. Compared to the DeepTriage baseline [8], our model achieved substantial improvements in recommendation accuracy, with gains of approximately 0.3579 in the best configuration and 0.1324 even in the worst. Statistical tests confirmed that these improvements are significant [9,10].

The main contributions of this study are as follows:

- **Developer-Specific Top-K Feature Selection:** We introduce a dynamic top-K feature extraction method that tailors feature selection to each developer's historical bug-fixing profile. This approach significantly reduces irrelevant information, enhances model precision, and shows superior performance over baseline and ablation models.
- **Robust Performance Across Projects:** Extensive experiments on datasets from Google Chrome, Mozilla Core, and Mozilla Firefox show that our model consistently outperforms DeepTriage across multiple metrics, including accuracy, precision, recall, and the F1-score, confirming its generalizability and robustness.
- **Practical Efficiency Gains:** By minimizing the need for reassignment, our model reduces bug resolution delays and maintenance costs. The proposed method offers a practical and scalable solution for improving developer assignment in large-scale open-source environments.

In summary, this study presents an automated developer recommendation method that enhances assignment accuracy through advanced feature selection and deep learning techniques. The experimental results show the method's potential to optimize bug triaging workflows, improve developer utilization, and elevate software quality in open-source ecosystems.

2. Background

Software plays a critical role across various fields, and bugs are an inevitable aspect of the software development lifecycle. The prompt resolution of these bugs is essential to ensure high-quality software and maintain user satisfaction. In open-source projects, however, approximately 50% of the bugs initially assigned to developers are reassigned due to mismatches in expertise or availability, leading to delays in resolution and increased software maintenance costs [1]. Addressing this issue by accurately assigning bug reports to the most suitable developers can significantly enhance software quality and streamline the maintenance process.

Figure 1 illustrates the typical bug correction workflow in open-source projects. In this process, bugs are identified and reported by end users, developers, or testers, often in textual format. These reports are submitted to the project repository, where they are reviewed by a project manager or team leader. At this stage, the manager manually analyzes the content of the reports to determine their nature and assigns them to developers based on their perceived expertise or current workload. While this manual approach ensures that every bug report is addressed, it often results in inefficiencies due to human error, subjective judgment, or an incomplete understanding of developer expertise.

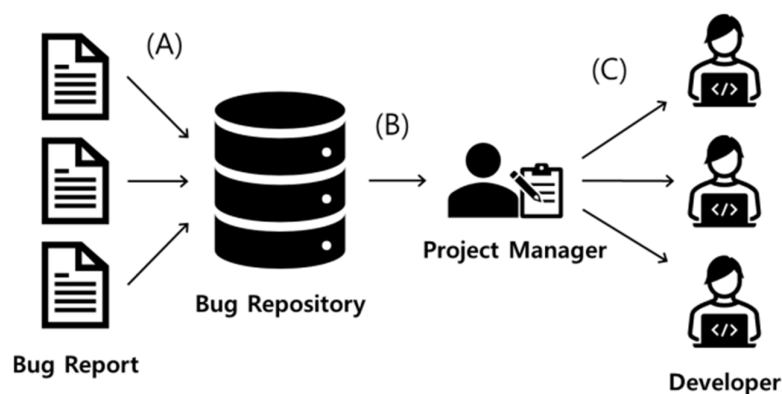


Figure 1. Process of developer recommendation in open-source projects. (A) A bug report is submitted by the user and stored in the bug repository. (B) The project manager reviews the submitted bug report. (C) The project manager assigns a suitable developer to address the reported issue.

Current bug reporting and tracking systems provide a structured framework for managing submitted bug reports but lack effective automation for identifying the most appropriate developer for each issue. This limitation underscores the need for more advanced tools to improve the assignment process. Machine learning-based developer recommendation systems offer a promising solution to this challenge. By leveraging historical bug data, analyzing developer expertise, and applying sophisticated algorithms, these systems can recommend the most qualified developer to address a specific bug. Such automation has the potential to reduce the frequency of reassignment, minimize resolution times, and lower the overall maintenance costs. Moreover, the implementation of these systems can improve the consistency and fairness of developer assignments, thereby enhancing the efficiency and outcomes of open-source software projects.

2.1. Bug Report

Bug reports are an integral part of the software maintenance process, providing essential documentation of issues or defects encountered within a software system. These reports are typically authored by end users, developers, or testers, allowing them to detail problems encountered during software usage or testing. While bug reports are often written in free-form text to allow reporters to describe issues comprehensively, the structure

and format can vary depending on the specific bug-tracking system employed by the open-source project [11]. Many modern bug-tracking systems offer customizable settings, enabling users to input relevant details in a structured or semi-structured manner, which enhances the clarity and utility of the reports.

Figure 2 illustrates an example of a bug report submitted for Google Chrome [12]. This report (ID #1358640), filed on 1 September 2022, by tmathmeyer@chromium.org, documents a defect in the D3D11VideoDecoder module. The issue was described as “AV1 film grain parameters are copied incorrectly,” pointing to an error in parameter passing within the “Video Decoder” module of the software. The responsibility for resolving this bug was assigned to jianlin.qiu@intel.com, who addressed the issue by modifying the affected components. This example underscores the typical workflow of bug reports, from initial submission to resolution by an appropriately assigned developer.

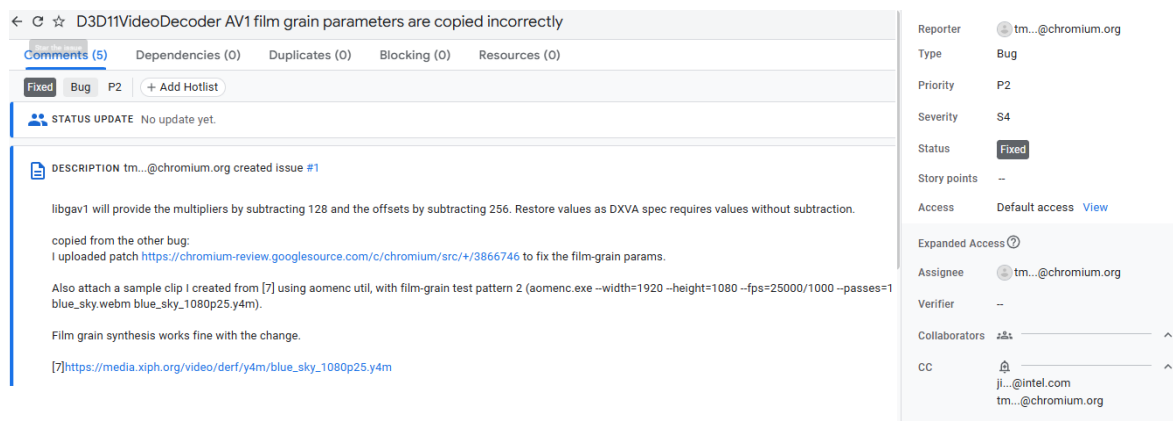


Figure 2. Example of Google Chrome bug report (#1358640).

Bug reports play a pivotal role in detecting and addressing software defects by serving as a primary source of information about problems encountered within a system. They typically include detailed descriptions of the issue, the affected components, and relevant technical details, which provide project managers and development teams with the necessary context to assess and resolve the problem. A thorough analysis of these reports enables managers to assign developers with the appropriate expertise, reducing the likelihood of re-assignment due to mismatched skills or responsibilities. By facilitating a more streamlined resolution process, the proper handling of bug reports minimizes delays and improves the overall efficiency of software maintenance workflows. This not only enhances the quality of the software but also reduces maintenance costs and increases user satisfaction.

2.2. Bug-Tracking System

To enhance efficiency in software development and maintenance, developers and teams increasingly rely on bug-tracking systems [11]. These systems serve as essential tools for managing bug reports, tracking project progress, and ensuring transparency and accountability throughout the development lifecycle. By centralizing bug-related information, they provide a unified platform where users can report issues and developers can monitor, prioritize, and address them systematically.

Figure 3 presents an example of a bug-tracking system utilized during the development of Google Chrome [13]. In this example, the search condition is set to “Fixed,” allowing users to filter and view the resolved issues. The interface prominently displays the essential attributes of each bug report, including its unique identification number, priority level, type, title, status, and the date of the last modification. Beyond these visible details, the system

also maintains extensive internal data, such as the historical context of the issue, the affected modules, associated discussions, and the developers who contributed to its resolution.

P	TYPE	TITLE
P2	Bug	 LLD undefined symbol error in some Rust executables when profiling instrumentation is enabled
P2	Bug	[Messages] Every time you open the app it'll force close and give the app crashed error
P2	Bug	 [Build] - Android build fails on dockerized Ubuntu
P1	Bug	 Tests are failing: devtools_agent_coverage_observer.cc(122): Check failed: !base::Contains(devtools_agents_, host).
P2	Feature Request	Tasks API not importing time from Google Sheets
P1	Bug	 gl_tests_validating failing on Pixel 2/4 bots
P2	Bug	Web App IFrame fails to initialize with TypeError
P2	Bug	 Setting custom user-data-directory AND chromedriver binary path arguments causes Chrome error
P2	Bug	 Broken build with safe_browsing_mode==0 on linux
P2	Bug	[Gemini] Selecting the Gemini icon in Google Messages crashes app and reverts to home screen.
P2	Bug	 "Warning - Can't connect firewall to HTTPS websites from DNS Android apps" is seen when connecting firewall to HTTPS websites in
P2	Bug	Apps Script web app TypeError: vb.X is not a constructor

Figure 3. Example of bug-tracking system (Google Chrome).

The Google Chrome bug-tracking system offers advanced features that enhance its functionality, such as refined search options, customizable data displays, and tools to prioritize critical bugs effectively. These features allow users to filter and analyze bug-related data with precision, ensuring that high-priority issues receive prompt attention. The system also facilitates tracking the progress of specific bugs over time, enabling project managers and teams to monitor ongoing work and measure resolution efficiency.

Bug-tracking systems are particularly valuable in open-source projects, where the sheer volume of reported bugs can overwhelm manual processes. By organizing and presenting data in a structured format, these systems help developers focus on the most urgent and impactful issues, reducing delays in resolution and improving overall software quality. Additionally, their ability to store and analyze historical data contributes to long-term improvements in project management practices, fostering better decision-making and accountability. These benefits make bug-tracking systems indispensable tools for maintaining the reliability, efficiency, and user satisfaction of modern software projects.

Effective software maintenance relies heavily on accurate developer recommendations, and this study proposes a machine learning-based approach to address this need. This chapter underscores the significance of defect management in software development and highlights the necessity for precise developer assignment. It reviews the limitations of current manual and automated bug-tracking systems, emphasizing the challenges posed by inefficiencies and reassignments. Subsequent sections introduce the proposed methodology, which combines a developer-specific top-K feature selection algorithm with a hybrid CNN-LSTM model, designed to enhance recommendation accuracy. The experimental analysis shows that the proposed model significantly outperforms existing approaches, delivering higher precision and efficiency in developer assignments. These improvements lead to enhanced software quality, reduced maintenance costs, and optimized workflows, underscoring the practical and technical contributions of this research.

3. Related Work

Significant research has explored the optimization of developer recommendations and bug classification systems, primarily leveraging machine learning techniques to enhance assignment accuracy. Anvik et al. [14] showed the efficacy of text classification for

automating bug report assignments, achieving results comparable to naïve Bayes classifiers. This foundational work inspired subsequent studies like Xuan [15], who introduced a semi-supervised text classification approach combining expectation maximization and Bayesian methods to iteratively annotate unlabeled data and recommend developers. While these studies underscore the potential of text classification in improving developer assignment, their reliance on static approaches limits their adaptability to dynamic project environments.

Expanding on this, Ge et al. [16] integrated feature and instance selection to enhance dataset quality, emphasizing developer participation in bug classification. Similarly, Yadav et al. [17] proposed a two-step approach that constructed developer profiles based on contributions and performance metrics. These approaches highlighted the importance of tailored data preprocessing and developer profiling but lacked robust mechanisms to address scalability and cross-domain applicability.

Further advancements incorporated historical data and temporal dynamics into recommendation systems. Xia et al. [18] introduced DevRec, which analyzed past bug reports and developer activities to improve recommendation precision, while Shokripour et al. [19] utilized term-weighting techniques incorporating temporal information. Xi et al. [20] developed DeepTriage, leveraging developer persistence to enhance classification accuracy. However, these methods often fell short in addressing noise in the data and achieving broad generalizability.

Recent deep learning approaches have pushed the boundaries of developer recommendation systems. Zaidi et al. [21] utilized CNNs with advanced embeddings like word2vec, GloVe, and ELMo, showing that ELMo embeddings yielded the best performance. Mian et al. [22] employed a bi-LSTM-DA model for robust word representations, and Liu et al. [23] combined BERT-based textual analysis with heterogeneous collaborative networks to improve recommendation accuracy. These methods showcased the power of deep learning in capturing complex patterns but were often computationally intensive and constrained by dataset-specific configurations.

Wang et al. [24] proposed a supervised contrastive learning approach that leverages the similarity between the historical bug repair records of developers. By comparing the repair histories of similar developers within the same batch to those of dissimilar developers, their method demonstrated an improved performance in adversarial settings and with limited training data across various architectures, including Bi-LSTM + ELMo, Bi-LSTM-A + ELMo, and BERT.

Tian et al. [25] introduced a model that integrates a developer's historical activity data with the suspicious code locations related to bug reports, fusing these features in a unified representation. This model significantly outperformed both location-based and activity-based approaches across several open-source projects, including Eclipse JDT, Eclipse SWT, and ArgoUML, confirming its effectiveness.

Liu et al. [26] proposed a bug triage method called MCNN-BT, which combines word embeddings with a multi-scale convolutional neural network to enhance the extraction of features from bug report texts. Experimental results showed that this method substantially outperformed traditional approaches like Naive Bayes and LDA across five open-source projects.

Kumar Dipongkor [27] employed six pre-trained large language models (LLMs), fine-tuning them for a sequence classification task tailored to error classification. He proposed a voting-based ensemble approach, which outperformed individual models in terms of classification accuracy.

Chhabra et al. [28] explored an automated bug triage approach using LLM embedding chains. Their method not only assigned bugs to appropriate developers but also predicted their priority levels.

Jahanshahi et al. [29] developed ADPTriage, a bug triage method based on a Markov Decision Process (MDP). This approach accounts for each developer's relevant expertise when assigning bugs in real time. Compared to myopic methods, ADPTriage significantly improved both assignment accuracy and bug resolution time.

Unlike these prior efforts, this study introduces a dynamic top-K feature selection algorithm that prioritizes developer-specific features from bug reports, reducing noise and enhancing model precision. By integrating a hybrid CNN-LSTM architecture, the proposed model leverages the spatial feature extraction capabilities of CNNs and the sequential dependency modeling strengths of LSTMs. This approach captures nuanced patterns in bug reports more effectively than standalone models.

Additionally, this study addresses scalability and domain adaptability by emphasizing cross-domain evaluation. Unlike earlier research focused solely on open-source or business projects, the proposed methodology bridges these domains, showing its applicability in diverse settings. Rigorous statistical validation, including *t*-tests and Wilcoxon tests, further establishes the reliability of the observed performance improvements, a level of scrutiny often missing in prior work.

Finally, the large-scale application of the proposed model to over 850,000 bug reports involving approximately 3900 developers highlights its scalability and practical utility. By addressing limitations in feature selection, adaptability, and computational efficiency, this study offers a robust framework for improving developer recommendation systems, paving the way for future advancements in this field.

4. Developer Recommendation Methodology

Figure 4 presents a schematic overview of the proposed developer recommendation method. The primary objective of this approach is to accurately recommend the most suitable developer for resolving a given bug report, thereby minimizing the reassignment rates and improving bug resolution efficiency.

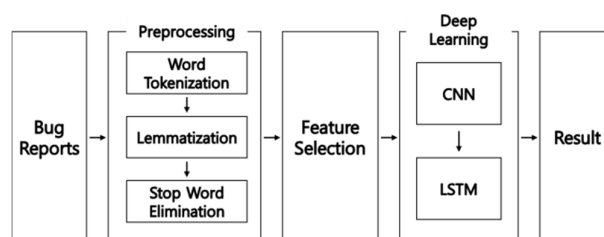


Figure 4. Schematic of proposed method.

The process begins with the extraction of bug reports associated with each developer from the bug repository. These reports are subjected to preprocessing steps to clean and standardize the data, ensuring that it is suitable for analysis. Next, a feature selection algorithm [6] is employed to extract the relevant features for each developer. These features capture critical information about the developer's expertise, historical bug resolution patterns, and other relevant attributes. The extracted developer-specific features are then used as input to a hybrid CNN-LSTM algorithm [7]. The CNN component identifies spatial patterns in the data, while the LSTM component captures sequential dependencies and temporal patterns in the bug reports. By combining these two approaches, the model is able to make nuanced and accurate developer recommendations. The final output of the method is a recommendation for the most appropriate developer to address a specific bug

report. This automated approach reduces the reliance on manual assignments by project managers, streamlines the bug resolution process, and enhances the overall efficiency and quality of software maintenance.

This method provides a robust and scalable solution for developer assignment in open-source projects, addressing the challenges of high reassignment rates and increasing software maintenance costs.

4.1. Preprocessing

Bug reports are typically written in free-text format, allowing users to describe issues in their own words. To prepare these reports for analysis, they undergo preprocessing [8], which includes several key steps, such as tokenization, lemmatization, and stop-word removal. These steps standardize the text data, making it suitable for machine learning algorithms and semantic analysis.

- **Tokenization:** This process involves breaking the text into individual tokens, typically words or phrases, to create a structured representation of the report. For example, sentences are decomposed into their constituent words, which are then extracted for further processing.
- **Lemmatization:** During this step, each word is reduced to its base or root form. This normalization ensures that variations in the same word are treated uniformly. For instance, words such as “notes,” “sounding,” and “heights” are converted to their root forms, “note,” “sound,” and “height,” respectively. By standardizing word forms, lemmatization helps to reduce noise in the dataset.
- **Stop-Word Removal:** Commonly used words that carry little to no semantic meaning, such as “is,” “me,” “over,” and suffixes, are removed. Eliminating these stop-words reduces the dimensionality of the data and focuses the analysis on the meaningful terms relevant to the bug descriptions.

After preprocessing, the bug reports are systematically extracted from the repository for each developer. This step ensures that the data is standardized and cleaned, providing consistent and high-quality input for further analysis. Preprocessing eliminates redundancies, filters out irrelevant information, and organizes the reports into a structured format, which is critical for the subsequent stages of feature selection and machine learning. By focusing on the most pertinent aspects of the bug reports, preprocessing not only enhances the relevance of the input data but also ensures that the model operates efficiently and effectively.

4.2. Feature Selection Algorithm

Following preprocessing, the next step involves identifying and selecting the relevant features from the bug reports. This process is essential for reducing the input dimensionality, minimizing noise, and improving the effectiveness of the learning model. To accomplish this, we apply a developer-specific top-K feature selection algorithm, as illustrated in Figure 5. This algorithm extracts the most informative words from each bug report based on their historical relevance to individual developers.

Each word in a bug report is assigned a relevance score that reflects how frequently and distinctively the word has appeared in previous bug reports handled by a particular developer. Words that are frequently used by one developer but rarely used by others are considered more representative of that developer’s bug-fixing behavior. Based on these scores, the algorithm selects the top-K words with the highest relevance for each developer. The selected features are then used as input to the CNN-LSTM model.

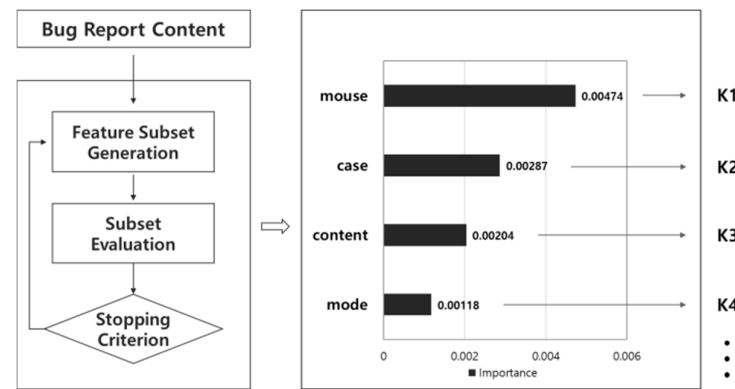


Figure 5. Feature selection for top-k extraction.

To assess the impact of feature selection, we varied the value of K from 1 to 20:

- Top-1 selects the single most relevant word;
- Top-2 includes the two highest-ranked words;
- Top-20 includes the twenty most relevant words per developer.

By focusing on highly relevant terms, this approach enables the model to emphasize developer-specific patterns, thereby improving both the training efficiency and recommendation accuracy.

We note that selecting only the top-K features may result in the exclusion of low-frequency but semantically important terms, particularly in the case of specialized bug-fixing domains. To mitigate this, our scoring approach incorporates both statistical frequency and contextual relevance, helping to preserve domain-specific terminology that may be critical despite its rarity.

The decision to use a K range between 1 and 20 was guided by the empirical analysis, as discussed in Section 5.5.1. Our ablation experiments demonstrated that the model accuracy generally improved as K increased, particularly up to $K = 15$. Beyond that point, performance gains plateaued, indicating diminishing returns. This result supports the chosen K range as a balanced trade-off between informativeness and model complexity.

In summary, the top-K feature selection algorithm provides a targeted, developer-aware method for identifying meaningful textual inputs. It enhances the model's interpretability and performance while maintaining computational efficiency.

4.3. CNN-LSTM Algorithm

To generate personalized developer recommendations for new bug reports, we train a hybrid deep learning model that combines a convolutional neural network (CNN) with a Long Short-Term Memory (LSTM) network. Figure 6 provides an overview of the architecture.

The CNN component is designed to process high-dimensional textual feature inputs while preserving spatial relationships between terms. It consists of multiple convolutional layers that extract local patterns from the input sequences. To prevent an excessive reduction in feature map size due to kernel and stride operations, padding is applied to maintain spatial resolution. To further enhance computational efficiency, max pooling layers are introduced to discard less informative activations, allowing the network to retain only the most salient features.

The output of the CNN is then passed to the LSTM layer, which models the temporal dependencies among the extracted features. Unlike traditional recurrent neural networks (RNNs), LSTMs are capable of retaining long-term dependencies and selectively remembering or forgetting past information. This makes them particularly effective for processing sequential data, such as the structured and context-rich representations of bug reports.

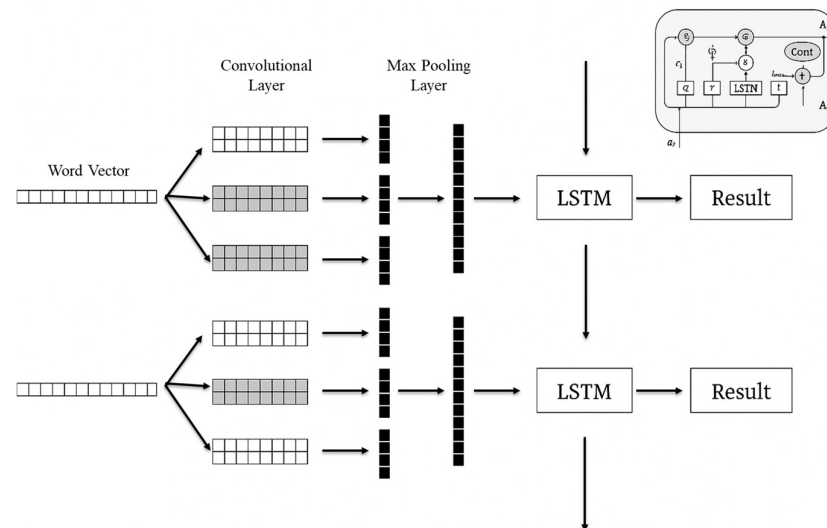


Figure 6. Schematic of CNN-LSTM algorithm.

In this study, developer-specific top-K textual features are first selected through the feature selection algorithm described in Section 4.2. These features are then encoded and passed through the CNN, followed by the LSTM, to generate a ranked recommendation of developers for each bug report. The value of K is varied during training to identify the optimal number of features that maximize recommendation accuracy.

By integrating the CNN and LSTM architectures, the model is able to capture both spatial and sequential patterns in bug report data. This combined approach improves learning effectiveness, enhances personalization, and delivers accurate developer recommendations across diverse open-source project contexts.

5. Experimental Analysis

In this study, bug reports were extracted from the bug repository, and a feature selection algorithm was applied to identify the relevant features for each developer. From the extracted features, the top-K feature words were selected and processed through the CNN-LSTM model. The CNN was used to analyze the spatial patterns of the features, and its output was subsequently passed into the LSTM, which captured sequential and temporal dependencies. The final output of the model was a recommendation for the most suitable developer for the given bug report. Figure 7 illustrates the configuration of the CNN-LSTM algorithm.

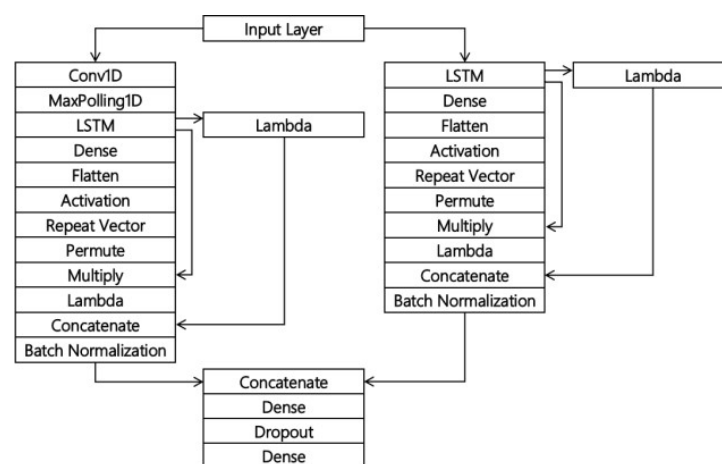


Figure 7. CNN-LSTM model summary.

The proposed model features a hybrid architecture combining the CNN and LSTM layers, specifically designed to handle high-dimensional and sequential data effectively. The CNN component extracts critical spatial features from the input data, while the LSTM component captures temporal relationships among the extracted features, enabling the model to generate accurate developer recommendations. This architecture is particularly well-suited for processing bug report data, which often contains structured and sequential patterns.

The hyperparameters of the CNN-LSTM model were carefully selected to optimize both performance and computational efficiency. The Adam optimizer was utilized due to its robust optimization capabilities, including adaptive learning rates that improve convergence speed while maintaining training stability. A learning rate of 1×10^{-4} was chosen, striking a balance between rapid convergence and avoiding overshooting during optimization. To address the multi-class classification nature of the developer recommendation task, the model employed the categorical cross-entropy loss function. This loss function measures the divergence between predicted and actual class probabilities, driving the model toward accurate predictions across multiple classes.

The CNN component includes a Conv1D layer configured with 128 filters, which enables the extraction of detailed spatial features from the input data. The use of the ReLU (Rectified Linear Unit) activation function introduces non-linearity, enhancing the model's ability to learn complex patterns and mitigating the vanishing gradient problem often encountered in deep architectures. This configuration ensures effective feature extraction, which plays a vital role in the model's overall performance.

These hyperparameter choices were tailored to support the model's ability to process high-dimensional data efficiently while maintaining high predictive accuracy. The Adam optimizer and the categorical cross-entropy loss function worked in tandem to optimize the model's learning process, while the Conv1D layer with 128 filters and ReLU activation contributed to robust feature extraction.

Through this configuration, the CNN-LSTM model showed its capacity to process bug report data and recommend developers with high accuracy. Additionally, the integration of top-K feature selection further enhanced the model's effectiveness by ensuring that the most relevant features were utilized during training. The combination of a carefully designed architecture, optimized hyperparameters, and effective feature selection contributed to the success of the experimental results, validating the model's utility in the context of developer recommendations.

5.1. Experimental Dataset

The dataset for this study was derived from three prominent open-source projects [8]: Google Chrome, Mozilla Core, and Mozilla Firefox. These projects were selected due to their extensive bug report histories and the diversity of their development teams, making them suitable candidates for evaluating the proposed developer recommendation method.

Table 1 provides an overview of the bug report data used in the experiments. The Google Chrome dataset includes bug reports collected between August 2008 and July 2013, while the Mozilla Core dataset spans from March 1997 to June 2016, and the Mozilla Firefox dataset covers the period from July 1998 to June 2016. In total, the combined dataset consists of 859,799 bug reports.

Each dataset includes bug descriptions, timestamps, severity levels, and developer assignment records, with approximately 3900 unique developers across all projects. These developers represent distinct classes in the multi-class classification task performed by the model.

To ensure fair and consistent evaluation, each dataset was randomly divided into three subsets with a ratio of 8:1:1 for training, validation, and testing, respectively. The

training set was used to optimize the model parameters, the validation set was employed for hyperparameter tuning and early stopping, and the testing set was reserved for final performance evaluation. This split strategy allows for an objective assessment of the model's generalization capability across unseen data.

Table 1. Summary of datasets.

	Google Chromium [8]	Mozilla Core [8]	Mozilla Firefox [8]
# of Reports	383,104	314,388	162,307
# of Developers	1944	1375	581

The diversity and volume of the datasets, along with the inclusion of both the structured and unstructured features, provide a robust foundation for training and evaluating the proposed method. Furthermore, the use of multiple real-world projects helps show the method's scalability and applicability in different open-source environments.

While the selected datasets from Google Chrome, Mozilla Core, and Mozilla Firefox represent well-established open-source systems with large developer bases, they primarily reflect desktop software ecosystems. As such, the current evaluation may not fully capture the characteristics of other types of software systems, such as mobile applications, domain-specific libraries, or enterprise software. Extending the evaluation to include more diverse project types will be a valuable direction for future work to further validate the generalizability of the proposed method.

5.2. Research Questions

This study was conducted to evaluate the efficiency and applicability of the proposed developer recommendation algorithm. The following research questions guided the experimental analysis:

RQ1: Does the proposed feature extraction method improve the accuracy of developer recommendations?

To determine the effectiveness of the proposed model, its performance must first be evaluated independently and compared with the baseline and other relevant methods. By systematically adjusting the top-K parameters in the feature selection process, the optimal configuration for developer recommendations can be identified. This ensures that the feature extraction method is capable of selecting relevant attributes that significantly contribute to the model's accuracy. The performance of developer recommendations is then verified through comprehensive testing.

RQ2: Can the proposed model be effectively applied to developer recommendation systems?

The applicability of the proposed model is assessed by comparing its performance with the baseline (DeepTriage) and other related studies. If the proposed model consistently outperforms these alternatives, it shows its potential for integration into practical developer recommendation systems. Statistical analysis [9,10] is used to confirm whether the observed performance differences between the proposed model and the baseline are significant. This validation establishes the reliability and robustness of the proposed model for real-world applications.

By addressing these research questions, this study aims to show the practical advantages of the proposed algorithm in recommending developers effectively while validating its performance against established benchmarks.

5.3. Evaluation Metrics

To assess the performance of the proposed model, a subset of ML evaluation metrics was employed [30,31], specifically, precision, recall, F1-score, and accuracy. These metrics are computed using Equations (1)–(4), respectively.

$$\text{Precision} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Positive}} \quad (1)$$

Precision measures the ratio of correctly identified developer recommendations to all positive predictions, highlighting the model's ability to minimize false recommendations.

$$\text{Recall} = \frac{\text{TP True Positive}}{\text{True Positive} + \text{False Negative}} \quad (2)$$

Recall evaluates the model's capability to identify all relevant developers, ensuring comprehensive coverage for bug reports.

$$\text{F1}_{\text{score}} = \frac{2 * \text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}} \quad (3)$$

The F1-score provides a harmonic mean of precision and recall, offering a balanced metric that is particularly useful for imbalanced datasets.

$$\text{Accuracy} = \frac{\# \text{ of Correct Predictions}}{\text{Total of Predictions}} \quad (4)$$

Accuracy calculates the overall proportion of correct developer recommendations across all bug reports.

For example, if the model correctly assigns developers for 75,516 out of 100,000 bug reports, the accuracy metric would yield a value of 75.52%. This indicates that the model successfully recommended the appropriate developer in approximately three-quarters of the cases.

These evaluation metrics collectively provide a comprehensive assessment of the model's performance, capturing its precision in making correct recommendations, its ability to identify all relevant developers, and its overall accuracy across the dataset.

5.4. Baseline

The proposed model is designed to recommend the most appropriate developers for a given bug report. To evaluate its effectiveness, the model's performance was compared with relevant baseline approaches. In this study, DeepTriage [8] was selected as the primary baseline.

DeepTriage is a publicly available developer recommendation model that employs a deep bidirectional recurrent neural network (Bi-RNN) to predict the most suitable developer for a given bug report. It was selected because its implementation and datasets are openly accessible, allowing for full experimental replication under consistent conditions.

While several other developer recommendation methods have been introduced in previous studies, such as DevRec, DRETOM, and BERT-based models, most of these do not provide a publicly available source code or use datasets that are not openly accessible. Consequently, the direct reproduction of these methods was not feasible, making fair empirical comparison difficult. In contrast, DeepTriage offers a reproducible and widely cited baseline, which makes it a suitable benchmark for evaluating the proposed method.

In addition to DeepTriage, we implemented several traditional machine learning algorithms commonly used in prior work on bug triaging. These included Naive Bayes, BayesNet, and J48. All models were trained and evaluated using the same features and

experimental settings. This approach ensures a comprehensive and objective evaluation of the proposed method using both state-of-the-art and classical baselines under reproducible experimental conditions.

5.5. Experimental Results

5.5.1. Results

The effectiveness of the top-K feature selection algorithm for each developer was validated before evaluating the overall performance of the proposed method. Figure 8 illustrates a performance comparison between applying the top-K feature selection algorithm and a non-feature selection approach. The accuracy for developer recommendations improved significantly when using the top-K algorithm across all datasets.

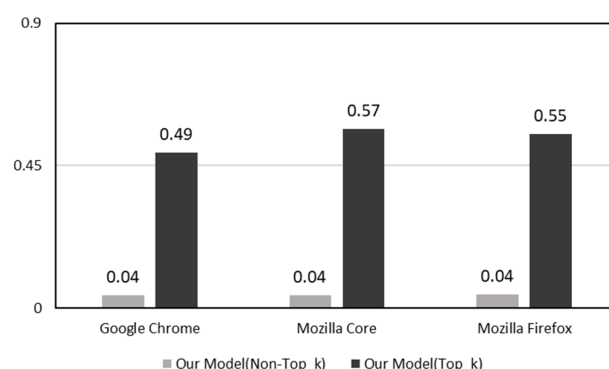


Figure 8. Performance comparison with and without top-k.

For the Google Chrome dataset, the accuracy increased from 0.04 in the non-feature selection approach to 0.49 with top-K feature selection, reflecting an improvement of approximately 0.45. In the Mozilla Core dataset, the accuracy improved from 0.04 to 0.57, showing a performance gain of 0.53. Similarly, the Mozilla Firefox dataset recorded an increase from 0.04 to 0.55, indicating an improvement of 0.51.

These results confirm that the top-K feature selection algorithm enhances the model's accuracy by prioritizing the most relevant features for each developer and minimizing noise. The consistent improvement observed across all datasets shows the robustness and effectiveness of this approach, justifying its incorporation into the proposed method.

The proposed model extracts developer-specific features and trains the recommendation system using the top-K feature words. Figure 9 depicts the impact of varying K values on the model's accuracy. The X-axis represents the value of K (the number of selected features), while the Y-axis shows the accuracy of the model's developer recommendations.

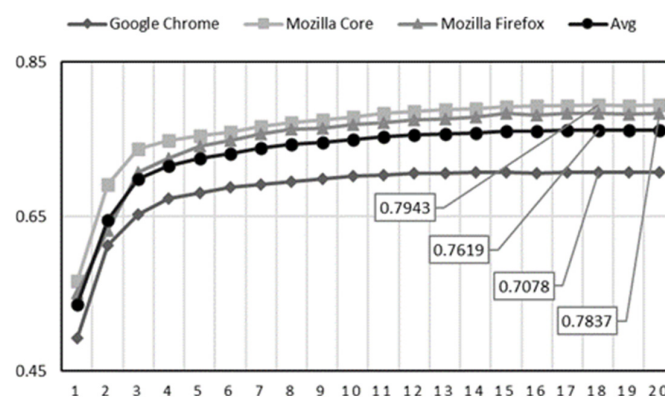


Figure 9. Results of performance with our model.

The results indicate a consistent improvement in performance as K increases. For the Google Chrome and Mozilla Core datasets, the best accuracy was achieved at $K = 18$, while the Mozilla Firefox dataset reached its optimal performance at $K = 20$. The overall average accuracy also shows a steady increase, peaking at 0.7837 for $K = 20$.

Notably, the performance gains are most pronounced when K is increased from 1 to 10, where the accuracy improves significantly for all datasets. Beyond $K = 10$, the rate of improvement diminishes, with performance differences becoming negligible as K approaches higher values. This plateau effect suggests that the inclusion of additional features beyond a certain threshold contributes little to further enhancing the model's accuracy.

To maximize the model's effectiveness and ensure reliable developer recommendations, the highest K value ($K = 20$) was used, reflecting the optimal balance between performance and feature selection. These results validate the importance of carefully selecting K to achieve the best possible outcomes in developer assignment tasks across diverse datasets.

Further experiments analyzed the relationship between the number of bug reports assigned to developers and recommendation performance. Classifiers were defined as follows:

- Classifier 5: Developers with at least five assigned bug reports.
- Classifier 10: Developers with at least ten assigned bug reports.
- Classifier 20: Developers with at least twenty assigned bug reports.

Figures 10–12 illustrate the relationship between the value of K and the accuracy of developer recommendations for the Google Chrome, Mozilla Core, and Mozilla Firefox datasets, respectively. The evaluation considers three classifier configurations, Classifier 5, Classifier 10, and Classifier 20, representing different thresholds for the number of bug reports assigned to the developers.

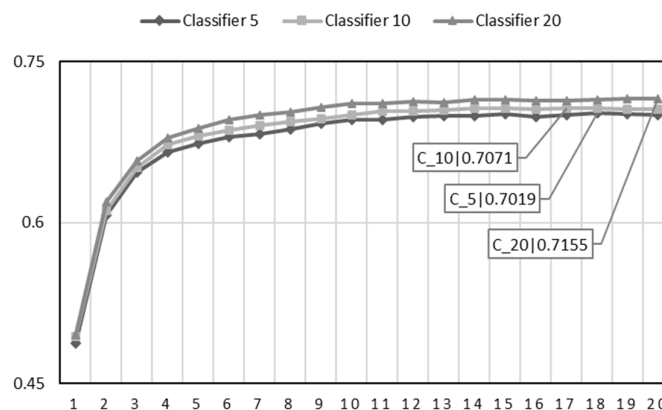


Figure 10. Performance comparison of each classifier for Google Chrome.

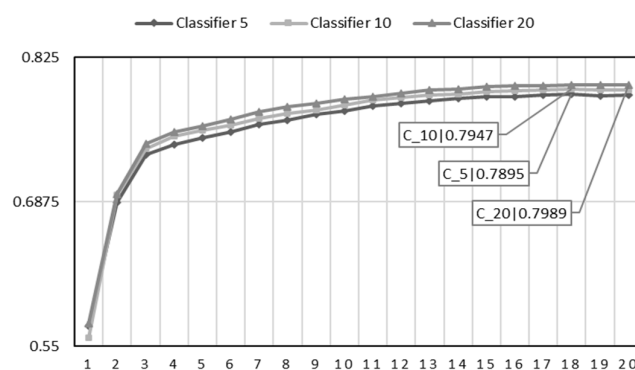


Figure 11. Performance comparison of each classifier for Mozilla Core.

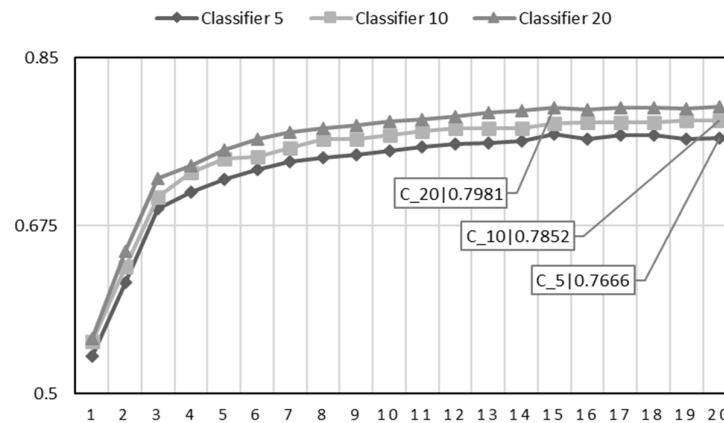


Figure 12. Performance comparison of each classifier for Mozilla Firefox.

In Figure 10, which depicts the results for the Google Chrome dataset, the accuracy consistently increased as K grew. The highest accuracy was observed at $K = 20$, where Classifier 20 achieved 0.7155, Classifier 10 achieved 0.7071, and Classifier 5 achieved 0.7019. The most significant improvements occurred between $K = 1$ and $K = 10$, after which the performance gains began to plateau. The differences in accuracy between classifiers became less pronounced as K increased, suggesting that the inclusion of additional features beyond $K = 10$ had a limited impact on performance. Figure 11, representing the Mozilla Core dataset, showed a similar trend. Accuracy improved steadily with increasing K , peaking at $K = 20$. At this value, Classifier 20 reached an accuracy of 0.7989, Classifier 10 achieved 0.7947, and Classifier 5 achieved 0.7895. As with the Google Chrome dataset, the most notable improvements were seen at lower K values, with performance stabilizing beyond $K = 10$. This indicates that the inclusion of up to 10 features was particularly beneficial, while additional features beyond this threshold contributed minimally to further accuracy improvements. In Figure 12, corresponding to the Mozilla Firefox dataset, the model's accuracy increased with K and reached its maximum at $K = 20$. Classifier 20 achieved 0.7981, while Classifier 10 and Classifier 5 reached 0.7852 and 0.7666, respectively. The performance gap between classifiers was more evident at lower K values but diminished as K increased, especially beyond $K = 10$.

Across all three datasets, the results showed that the accuracy of developer recommendations improved with higher K values, with the most substantial gains occurring at lower K values. The plateau effect observed beyond $K = 10$ indicates that additional features contributed less to performance improvements after this point. Classifier 20 consistently achieved higher accuracy compared to Classifier 10 and Classifier 5, showing that using more assigned bug reports per developer improved the model's performance. These findings highlighted the effectiveness of the proposed method across different datasets and classifier configurations.

Figures 13–15 illustrate the precision, recall, and F1-score metrics for the Google Chrome, Mozilla Core, and Mozilla Firefox datasets, respectively, as the value of K increases. The X-axis represents the value of K , while the Y-axis indicates the respective metric values. Across all datasets, the performance metrics show an increasing trend as K grows, highlighting the impact of feature selection on the model's performance.

In Figure 13, which corresponds to the Google Chrome dataset, the highest F1-score of 0.7824 is achieved at $K = 15$. Precision and recall also improve with increasing K , but they exhibit different growth rates. Precision reaches 0.5936, while recall achieves 0.6690 at $K = 15$. The performance stabilizes beyond $K = 10$, indicating that additional features contribute minimally to performance improvements after this threshold.

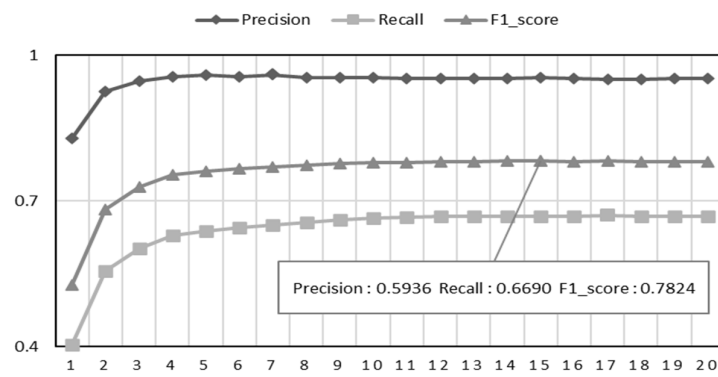


Figure 13. Performance of Google Chrome.

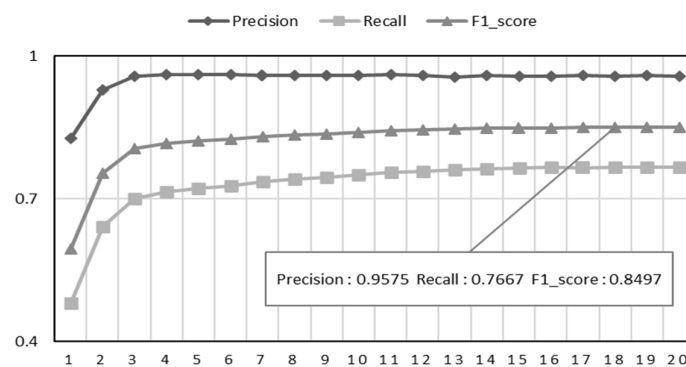


Figure 14. Performance of Mozilla Core.

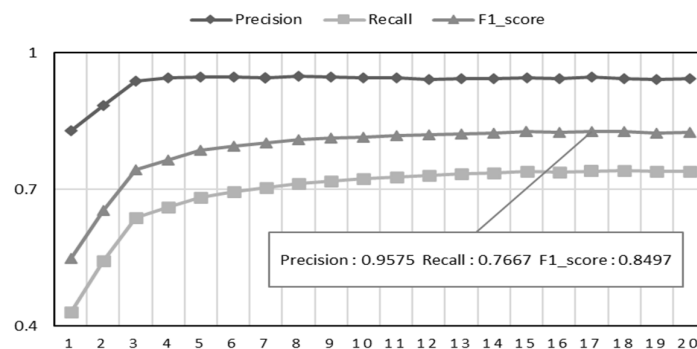


Figure 15. Performance of Mozilla Firefox.

Figure 14, representing the Mozilla Core dataset, shows that the highest F1-score of 0.8497 is observed at $K = 18$. Precision reaches a peak of 0.9575, while recall stabilizes at 0.7667. As with the Google Chrome dataset, the most significant improvements occur at lower K values, with performance gains diminishing beyond $K = 10$. This pattern suggests that selecting up to 10 features is sufficient for capturing the most relevant information for this dataset.

In Figure 15, which corresponds to the Mozilla Firefox dataset, the F1-score achieves its highest value of 0.8497 at $K = 17$. Precision reaches 0.9575, while recall stabilizes at 0.7667. The trends observed here are consistent with those in the other datasets, with substantial improvements at lower K values and minimal changes beyond $K = 10$.

The results across all three datasets indicate a general trend, where increasing the value of K improves precision, recall, and the F1-score. Notably, the most substantial performance gains are observed at lower K values, reflecting a diminishing return effect as K increases. The optimal K values, as indicated by the peak F1-scores, vary across datasets: $K = 15$ for Google Chrome, $K = 18$ for Mozilla Core, and $K = 17$ for Mozilla Firefox. This variability underscores the dataset-specific nature of feature selection, suggesting that the optimal K

value may depend on the intrinsic characteristics of the datasets, such as the distribution of features or the complexity of the underlying patterns.

In terms of the ROC-AUC scores [32], the highest K values yield scores of 0.6313 for Google Chrome and 0.8621 for both Mozilla Core and Mozilla Firefox. These metrics highlight the method's capability to maintain a balance between prediction accuracy and feature selection efficiency, particularly for datasets with differing levels of complexity.

The findings also reveal that while increasing K beyond a certain point can lead to further improvements in performance metrics, the benefits taper off, indicating a plateau effect. This diminishing marginal utility highlights the sensitivity of the model to the number of selected features, especially in the lower range of K. As part of our sensitivity analysis, we observed that accuracy improved substantially from $K = 1$ to $K = 15$, after which the gains diminished across all datasets.

This observation emphasizes the importance of balancing feature inclusion with computational efficiency. Overly high K values may yield marginal gains in accuracy while incurring greater computational overhead, which may not be practical for deployment in resource-constrained environments. Therefore, selecting an appropriate K value is not only critical for performance but also for ensuring real-world applicability. Future work could investigate adaptive or resource-aware strategies for K selection, enabling more robust and generalizable deployments across different project settings.

5.5.2. Comparison Results

Figure 16 compares the performance of the proposed developer recommendation model with the baseline DeepTriage across three open-source projects: Google Chrome, Mozilla Core, and Mozilla Firefox. The X-axis represents the projects, while the Y-axis denotes the accuracy of developer recommendations. The analysis evaluates two configurations of the proposed model, "Our Model (Best)" and "Our Model (Worst)," alongside the baseline DeepTriage.

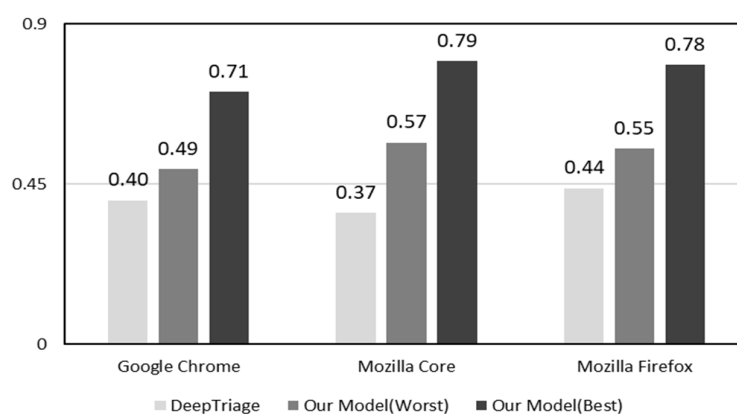


Figure 16. Performance comparison between models.

For the Google Chrome dataset, the proposed model in its best configuration ("Our Model (Best)") achieved an accuracy of 0.71, significantly outperforming the baseline DeepTriage, which achieved an accuracy of 0.40. Even in its worst configuration ("Our Model (Worst)"), the proposed model achieved an accuracy of 0.49, still higher than the baseline. This indicates that the proposed model consistently provides better developer recommendations compared to DeepTriage, regardless of configuration. In the Mozilla Core dataset, the proposed model also showed superior performance. The best configuration reached an accuracy of 0.79, while the baseline achieved only 0.37. The worst configuration of the proposed model achieved 0.57, highlighting a substantial improvement of 0.20 over the baseline, even under suboptimal conditions. Similarly, for the Mozilla Firefox dataset,

the best configuration of the proposed model recorded an accuracy of 0.78, significantly surpassing the baseline's accuracy of 0.44. The worst configuration achieved an accuracy of 0.55, showing a consistent improvement over the baseline by a margin of 0.11.

The results across all three datasets indicate that the proposed model, particularly in its optimal configuration, consistently outperforms DeepTriage. The improvement in accuracy for the best-performing configuration ranged from 0.34 to 0.42, depending on the dataset, while the worst-performing configuration still exhibited an improvement of approximately 0.11 to 0.20 over the baseline. These findings highlight the robustness and effectiveness of the proposed model in delivering accurate developer recommendations, even under less favorable configurations. The analysis shows the adaptability and reliability of the model across diverse open-source projects.

Figure 17 presents a comparative analysis of the proposed model against baseline methods and other machine learning and deep learning algorithms. The X-axis lists the different algorithms evaluated, while the Y-axis represents their corresponding accuracy percentages.

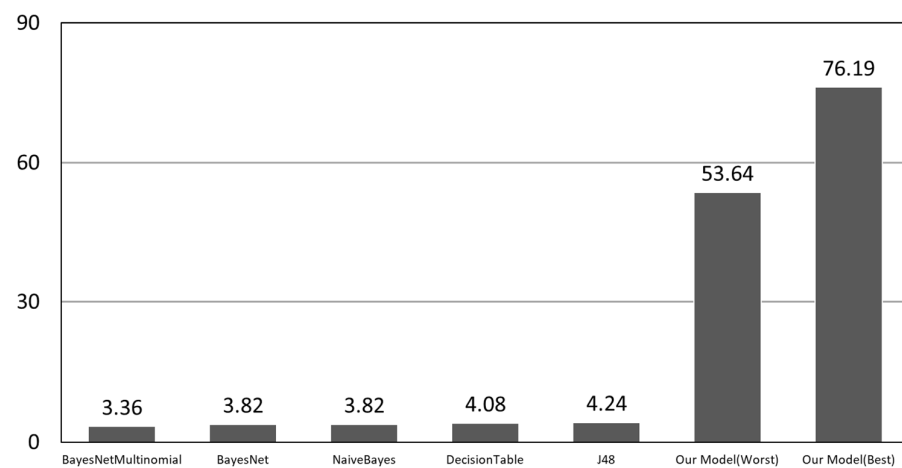


Figure 17. Performance comparison between machine and deep learning.

The proposed model shows a clear performance advantage over all other algorithms. In its best configuration ("Our Model (Best)"), the accuracy reaches 76.19%, significantly surpassing both the baseline and traditional machine learning methods. Even in its worst configuration ("Our Model (Worst)"), the accuracy of the proposed model is 53.64%, which is substantially higher than the other algorithms evaluated.

Among the traditional machine learning methods, J48 achieved the highest accuracy of 4.24%, followed by DecisionTable with 4.08%, BayesNet and NaiveBayes each with 3.82%, and BayesNetMultinomial with 3.36%. These results indicate that the traditional approaches struggled to provide effective developer recommendations, achieving performance levels far below the worst-performing configuration of the proposed model. The substantial difference in performance between the proposed model and the other algorithms highlights the advantages of its hybrid CNN-LSTM architecture and top-K feature selection approach. These elements allow the proposed model to capture complex patterns and dependencies in the data, which the other methods are unable to leverage effectively. The improvements in accuracy emphasize the efficacy of the proposed approach in addressing the limitations of the existing methods and achieving superior developer recommendation accuracy.

Additionally, statistical verification was performed [9,10] to evaluate the significance of performance differences between the proposed model and the baseline. The hypotheses tested are as follows:

The set null hypotheses are as follows:

- $H1_0$: No significant difference exists between the proposed model and DeepTriage for Google Chrome.
- $H2_0$: No significant difference exists between the proposed model and DeepTriage for Mozilla Core.
- $H3_0$: No significant difference exists between the proposed model and DeepTriage for Mozilla Firefox.

The alternative hypotheses against the established null hypotheses are as follows:

- $H1_a$: A significant difference exists between the proposed model and DeepTriage for Google Chrome.
- $H2_a$: A significant difference exists between the proposed model and DeepTriage for Mozilla Core.
- $H3_a$: A significant difference exists between the proposed model and DeepTriage for Mozilla Firefox.

For this study, “significant difference” was defined based on a threshold p -value of 0.05 in statistical testing. If the p -value is less than or equal to 0.05, the null hypothesis (H_0) is rejected in favor of the alternative hypothesis (H_a), indicating that the observed differences are statistically significant and not due to random variation.

To conduct the verification, a normality test [33] was applied using the F-measure. If the normal distribution’s p -value exceeded 0.05, a t -test was used [1,34]; otherwise, the Wilcoxon test was employed [1,34]. To ensure statistical rigor and fairness across models, we employed stratified 10-fold cross-validation. The dataset was partitioned into ten equal-sized folds while preserving the distribution of developer classes. For each fold, the model was trained on nine subsets and evaluated on the remaining one. This process was repeated ten times so that each fold was used exactly once as the test set. The F-measure values from each fold were recorded, and the resulting ten values were used as samples in the statistical significance tests. This approach ensured that all models were evaluated under consistent and balanced conditions.

Table 2 summarizes the results of the statistical tests.

Table 2. The statistical verification results.

Hypothesis	p -Value	Result
$H1_0$	(Wilcoxon test) 1.95×10^{-3}	$H1_a$: Accept
$H2_0$	(Wilcoxon test) 1.95×10^{-3}	$H2_a$: Accept
$H3_0$	(Wilcoxon test) 1.95×10^{-3}	$H3_a$: Accept

The statistical analysis revealed that the p -values for all the hypotheses were below the 0.05 threshold, indicating statistically significant differences in performance between the proposed method and the baseline model across all datasets. For example, in the case of the Google Chrome dataset, the p -value of 1.95×10^{-3} strongly supports the rejection of the null hypothesis and the acceptance of the alternative hypothesis. These results suggest that the proposed top-K feature extraction method consistently outperforms the baseline DeepTriage model.

To further validate the practical significance of these differences, we calculated the effect sizes using Cliff’s delta, which is appropriate for non-parametric comparisons. The effect size values were 0.77 for Google Chrome, 0.82 for Mozilla Core, and 0.79 for Mozilla Firefox. According to standard interpretation guidelines, values above 0.474 are considered “large” effects. These results indicate that the observed improvements are not only statistically significant but also substantial in magnitude.

Confidence intervals were also analyzed to assess the reliability of the observed performance gaps. The calculated intervals confirmed that the proposed method consistently delivered higher performance than the baseline, with limited overlap and strong central tendencies in all cases.

In summary, the combination of statistically significant p -values, large effect sizes, and narrow confidence intervals demonstrates that the proposed model offers meaningful and reliable improvements over the baseline. These findings support the conclusion that the top-K feature extraction approach significantly enhances developer recommendation performance across diverse open-source datasets, both statistically and practically.

6. Discussion

6.1. Results

The proposed model demonstrated superior performance in developer recommendation tasks compared to the baseline DeepTriage model. Significant improvements were observed across key evaluation metrics, including accuracy, precision, recall, and the F1-score. These results are primarily attributed to the integration of two components: the developer-specific top-K feature selection algorithm, which filters relevant features based on contextual relevance, and the hybrid CNN-LSTM architecture, which effectively captures both the spatial and sequential characteristics of bug report texts.

The quantitative analysis revealed a clear relationship between the number of selected features (K) and the model performance. As K increased from 1 to 10, the accuracy improved substantially across all datasets. For example, in the Mozilla Core dataset, the accuracy increased from 0.57 at $K = 1$ to 0.79 at $K = 20$. However, the rate of improvement diminished beyond $K = 15$, and additional features beyond this point often introduced redundancy. Moreover, higher K values were associated with increased training time and memory consumption. These observations underscore the importance of identifying an optimal K range that balances predictive performance with computational efficiency. Empirical evidence across datasets suggests that selecting K between 10 and 15 offers the best trade-off.

The role of feature selection was further validated by comparing the models trained with and without the top-K selection algorithm. The proposed method consistently outperformed the baseline across all datasets, achieving accuracy improvements ranging from 0.20 to 0.45. Features such as developer-associated keywords (e.g., function names or error types) contributed to this improvement by reducing noise and focusing learning on the most informative aspects of the bug reports.

The CNN-LSTM architecture also played a central role in performance gains. The CNN component effectively captured localized textual patterns, while the LSTM layer modeled long-term dependencies within bug reports. This hybrid structure enabled the model to learn more nuanced developer–bug relationships compared to models using the CNN or LSTM in isolation. Statistical significance testing further confirmed the robustness of the results, with all p -values falling below the 0.05 threshold when comparing the proposed model against the DeepTriage baseline.

From a practical perspective, the proposed model demonstrates potential for real-world deployment in large-scale open-source projects. By reducing the likelihood of developer reassignments, it contributes to more efficient bug triaging and resolution workflows. The model also maintained computational feasibility across experiments, suggesting that it can be integrated into existing development environments with minimal overhead.

Future research may focus on enhancing the adaptability of the feature selection process. Specifically, incorporating adaptive or data-driven strategies to dynamically select K based on dataset characteristics could further improve generalizability. In

addition, exploring the use of cross-lingual features, multi-lingual datasets, or industrial bug-tracking environments would broaden the applicability of the model beyond the open-source domain.

6.2. Threats and Validity

This study utilizes datasets from three large-scale open-source desktop software projects: Google Chrome, Mozilla Core, and Mozilla Firefox. Although these datasets provide extensive bug report histories and reflect mature development environments, they may not represent the full diversity of modern software systems. For example, bug reports from mobile applications, embedded systems, domain-specific libraries, or proprietary enterprise platforms often differ in format, terminology, and workflow structure. Therefore, the generalizability of the proposed method to such domains has not yet been validated. Future work should apply the model to a broader variety of software projects to assess its adaptability and robustness.

Another limitation involves the computational cost associated with the top-K feature selection process. Increasing the value of K generally leads to improved performance by including more relevant information. However, it also results in higher model complexity and longer training time. Selecting an appropriate value for K requires balancing predictive accuracy with computational efficiency. Although this study includes an analysis across a range of K values, further research could explore adaptive or data-driven methods to determine the optimal setting based on the characteristics of each dataset.

The model also relies heavily on historical bug-fixing data to identify developer-specific patterns. This dependence limits the system's ability to provide accurate recommendations for developers who are new or have limited prior activity. This issue, commonly referred to as the cold-start problem, is particularly relevant in dynamic environments where contributor turnover is frequent. Without sufficient historical data, the model may produce biased or incomplete recommendations. To address this, future work could incorporate additional developer-related information such as commit history, code ownership, or profile metadata. Alternative learning techniques, including transfer learning, dynamic embeddings, or zero-shot learning, may also help estimate developer expertise when historical data is unavailable.

Furthermore, the model does not consider dynamic or real-time information such as developer workload, recent activity, or ongoing task assignments. In real-world settings, assigning multiple tasks to high-performing developers without considering their availability can result in unbalanced workload distribution and decreased team productivity. Including time-based signals such as task queue length or the frequency of recent assignments could improve the fairness and realism of the recommendation process. Future research may combine static expertise modeling with real-time developer status to create more practical and team-aware assignment strategies.

The application of the model to proprietary or enterprise-level bug-tracking systems may also present challenges. These systems often differ in data structure, access control policies, and workflow mechanisms. To adapt the model for such settings, it would be necessary to design custom preprocessing components and adjust configurations to align with the specific operational constraints of each environment. This process would also provide an opportunity to test the model's scalability and effectiveness in complex and heterogeneous software development contexts.

Another challenge is the limited interpretability of the model. Because it uses deep learning components such as the CNN and LSTM, the decision-making process is not transparent. This lack of interpretability can hinder user trust and adoption in professional environments. Future improvements could involve the integration of explainable AI

techniques, such as attention-based visualization, SHAP, or LIME, to help users understand which features influence recommendations. In addition, case studies and decision-path visualizations could offer greater insights into how the model operates in specific examples.

Although the current study includes a sensitivity analysis for the top-K parameter, a further analysis of other hyperparameters such as CNN filter size, kernel width, LSTM units, and batch size could improve model robustness. It is also important to calibrate these parameters under practical conditions, including constraints related to computing resources, response latency, and training time. Such calibration would enhance the model's suitability for deployment in real-world development environments.

7. Conclusions

Software is widely used across various domains, and users frequently encounter bugs and suggest functional improvements. Ensuring a timely bug resolution is critical for maintaining high-quality software. However, in open-source projects, approximately 50% of bugs assigned to developers are reassigned due to incorrect fixes, leading to increased maintenance costs and inefficiencies. Addressing this challenge, the proposed study introduced a developer recommendation model that leverages feature extraction and a hybrid CNN-LSTM algorithm to assign the most appropriate developer for a given bug report.

The model incorporates a top-K feature selection algorithm, which identifies developer-specific features from bug reports to improve recommendation accuracy. The model's performance was rigorously evaluated and compared with the baseline DeepTriage method. Experimental results showed that the proposed model outperformed DeepTriage significantly. When tested on datasets from Google Chrome, Mozilla Core, and Mozilla Firefox, the best-performing configuration of the proposed model improved the accuracy by approximately 0.3579 compared to the baseline. Additionally, the worst-performing configuration of the proposed model still showed a difference of approximately 0.1324 over the baseline.

The statistical analysis confirmed that the performance differences between the proposed model and the baseline were statistically significant, validating the effectiveness of the top-K feature selection algorithm and the CNN-LSTM architecture. The results also highlighted a clear relationship between the value of K and the model's performance, with higher K values generally leading to better recommendations, until a plateau was reached. This finding underscores the importance of feature selection in improving developer recommendation systems.

This study shows that the proposed model effectively reduces reassignment rates and enhances the efficiency of bug resolution workflows in open-source projects. Future work may extend this research by conducting a comprehensive correlation analysis between the bug reports assigned to developers and the top-K values. This analysis will aim to further optimize the feature selection process and expand the applicability of the model to other domains, including proprietary and business projects.

Author Contributions: Software, J.J. and D.K.; Writing—original draft, G.Y. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by a research grant from Hankyong National University in the year of 2023.

Data Availability Statement: The original contributions presented in this study are included in the article. Further inquiries can be directed to the corresponding author.

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

References

1. Yang, G.; Zhang, T.; Lee, B. Towards semi-automatic bug triage and severity prediction based on topic model and multi-feature of bug reports. In Proceedings of the IEEE Annual Computer Software and Applications Conference, Vasteras, Sweden, 21–25 July 2014; IEEE Computer Society: Washington, DC, USA, 2014; pp. 97–106.
2. Jeong, G.; Kim, S.; Zimmermann, T. Improving bug triage with bug tossing graphs. In Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Amsterdam, The Netherlands, 24–28 August 2009; Association for Computing Machinery: New York, NY, USA, 2009; pp. 111–120.
3. Guo, S.; Zhang, X.; Yang, X.; Chen, R.; Guo, C.; Li, H.; Li, T. Developer activity motivated bug triaging: Via convolutional neural network. *J. Neural Process. Lett.* **2020**, *51*, 2589–2606. [\[CrossRef\]](#)
4. Jahanshahi, H.; Chhabra, K.; Cevik, M.; Başar, A. DABT: A dependency-aware bug triaging method. In Proceedings of the Evaluation and Assessment in Software Engineering, Trondheim, Norway, 21–23 June 2021; Chitchyan, R., Li, J., Eds.; Association for Computing Machinery: New York, NY, USA, 2021; pp. 221–230.
5. Park, J.W.; Lee, M.W.; Kim, J.; Hwang, S.W.; Kim, S. Costriage: A cost-aware triage algorithm for bug reporting systems. In Proceedings of the AAAI Conference on Artificial Intelligence, San Francisco, CA, USA, 7–11 August 2011; AAAI Press: Washington, DC, USA, 2011; Volume 25, pp. 139–144.
6. Ashokkumar, P.; Shankar, S.G.; Srivastava, G.; Maddikunta, P.K.; Gadekallu, R. A two-stage text feature selection algorithm for improving text classification. *ACM Trans. Asian Low-Resour. Lang. Inf. Process.* **2021**, *20*, 1–19.
7. She, X.; Zhang, D. Text classification based on hybrid CNN-LSTM hybrid model. In Proceedings of the International Symposium on Computational Intelligence and Design, Hangzhou, China, 8–9 December 2018; IEEE: New York, NY, USA, 2018; Volume 2, pp. 185–189.
8. Mani, S.; Sankaran, A.; Aralikkatte, R. DeepTriage: Exploring the effectiveness of deep learning for bug triaging. In Proceedings of the ACM India Joint International Conference on Data Science and Management of Data, New York, NY, USA, 3–5 January 2019; Association for Computing Machinery: New York, NY, USA, 2019; pp. 171–179.
9. The T-Testin Research Methods Knowledge Base. Available online: https://www.socialresearchmethods.net/kb/stat_t.php (accessed on 28 May 2022).
10. Wilcoxon, F. Individual comparisons by ranking methods. *Biom. Bull.* **1945**, *1*, 80–83. [\[CrossRef\]](#)
11. Zimmermann, T.; Premraj, R.; Sillito, J.; Breu, S. Improving bug tracking systems. In Proceedings of the International Conference on Software Engineering—Companion Volume, Vancouver, BC, Canada, 16–24 May 2009; IEEE: New York, NY, USA, 2009; pp. 247–250.
12. Bug Report. Google Chromium. Available online: <https://bugs.chromium.org/p/chromium/issues/detail?id=1358640> (accessed on 28 September 2022).
13. Google. Issue Tracker: Query Results for ‘Error’. Available online: <https://issuetracker.google.com/issues?q=error> (accessed on 16 January 2025).
14. Anvik, J.; Hiew, L.; Murphy, G.C. Who should fix this bug? In Proceedings of the International Conference on Software Engineering, Shanghai, China, 20–28 May 2006; Association for Computing Machinery: New York, NY, USA, 2006; pp. 361–370.
15. Xuan, J.; Jiang, H.; Ren, Z.; Yan, J.; Luo, Z. Automatic bug triage using semi-supervised text classification. *arXiv* **2017**, arXiv:1704.04769.
16. Ge, X.; Zheng, S.; Wang, J.; Li, H. High-dimensional hybrid data reduction for effective bug triage. *J. Math. Probl. Eng.* **2020**, *2020*, 5102897. [\[CrossRef\]](#)
17. Yadav, A.; Singh, S.K. A novel and improved developer rank algorithm for bug assignment. *J. Intell. Syst. Technol. Appl.* **2019**, *19*, 78–101. [\[CrossRef\]](#)
18. Xia, X.; Lo, D.; Wang, X.; Zhou, B. Accurate developer recommendation for bug resolution. In Proceedings of the IEEE 20th Working Conference on Reverse Engineering, Koblenz, Germany, 14–17 October 2013; IEEE: New York, NY, USA, 2013; pp. 72–81.
19. Shokripour, R.; Anvik, J.; Kasirun, Z.M.; Zamani, S. A time-based approach to automatic bug report assignment. *J. Syst. Softw.* **2015**, *102*, 109–122. [\[CrossRef\]](#)
20. Xi, S. DeepTriage: A bug report dispatching method based on cyclic neural network. *J. Softw.* **2018**, *29*, 2322–2335.
21. Zaidi, S.F.A.; Awan, F.M.; Lee, M.; Woo, H.; Lee, C.G. Applying convolutional neural networks with different word representation techniques to recommend bug fixers. *IEEE Access* **2020**, *8*, 213729–213747. [\[CrossRef\]](#)
22. Mian, T.S. Automation of bug-report allocation to developer using a deep learning algorithm. In Proceedings of the IEEE International Congress of Advanced Technology and Engineering, Taiz, Yemen, 4–5 July 2021; IEEE: New York, NY, USA, 2021; pp. 1–7.
23. Liu, B.; Zhang, L.; Liu, Z.; Jiang, J. Developer assignment method for software defects based on related issue prediction. *Mathematics* **2024**, *12*, 425. [\[CrossRef\]](#)
24. Wang, R.; Ji, X.; Tian, Y.; Xu, S.; Sun, X.; Jiang, S. Fixer-level supervised contrastive learning for bug assignment. *Empir. Softw. Eng* **2025**, *30*, 76. [\[CrossRef\]](#)

25. Tian, Y.; Wijedasa, D.; Lo, D.; Le Goues, C. Learning to rank for bug report assignee recommendation. In Proceedings of the IEEE International Conference on Program Comprehension (ICPC), Austin, TX, USA, 16–17 May 2016; IEEE: New York, NY, USA, 2016; pp. 1–10.
26. Liu, G.; Wang, X.; Zhang, X. Deep learning based on word vector for improving bug triage performance. In Proceedings of the international conference on Forthcoming Networks and Sustainability (FoNeS), Stevenage, UK, 3–5 October 2022; IET: London, UK, 2022; Volume 2022, pp. 770–775.
27. Dipongkor, A.K. An ensemble method for bug triaging using large language models. In Proceedings of the IEEE/ACM International Software Engineering: Companion (ICSE-Companion), Lisbon, Portugal, 14–20 April 2024; Association for Computing Machinery: New York, NY, USA, 2024; pp. 438–440.
28. Chhabra, D.; Chadha, R. Automatic bug triaging process: An enhanced machine learning approach through large language models. *Engineering. Technol. Appl. Sci. Res.* **2024**, *14*, 18557–18562. [[CrossRef](#)]
29. Jahanshahi, H.; Cevik, M.; Mousavi, K.; Başar, A. ADPTriage: Approximate dynamic programming for bug triage. *IEEE Trans. Softw. Eng.* **2023**, *49*, 4594–4609. [[CrossRef](#)]
30. Goutte, C.; Gaussier, E. A probabilistic interpretation of precision recall and F-Score with implication for evaluation. *Lect. Notes Comput. Sci.* **2005**, *3408*, 345–359.
31. Zhou, J.; Zhang, H.; Lo, D. Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. In Proceedings of the International Conference on Software Engineering, Zurich, Switzerland, 2–9 June 2012; IEEE: New York, NY, USA, 2012; pp. 14–24.
32. Hoo, Z.H.; Candlish, J.; Teare, D. What is an ROC curve? *Emerg. Med. J.* **2017**, *34*, 357–359. [[CrossRef](#)] [[PubMed](#)]
33. Shapiro–Wilk Test. Wikipedia. Available online: https://en.wikipedia.org/wiki/Shapiro-Wilk_test (accessed on 28 May 2022).
34. Shapiro, S.S.; Wilk, M.B. An analysis of variance test for normality (complete samples). *Biometrika* **1965**, *52*, 591–611. [[CrossRef](#)]

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.