# Bug Triaging based on Transformer Models Utilizing Commit Messages

AUTHORS NAME HIDDEN FOR ANONYMOUS REVIEW
Names anonymized
Location anonymized
Affiliation anonymized
Email anonymized

***Abstract*—Bug triaging is a crucial process in software maintenance involving assigning bug reports to the appropriate developers for resolution. Automated bug triaging eliminates the need for manual assignment by leveraging machine learning (ML), information retrieval (IR), and graph neural networks (GNN) techniques to classify bug reports based on extracted features. These approaches typically utilize bug report fields and metadata as training features. However, models often underperform when relying heavily on a few specific fields, such as the bug report summary and description. To address this issue, we integrated bug commit messages into the feature set to enhance the efficiency of bug-triaging models. Our study demonstrates significant improvements in triaging accuracy by incorporating commit messages. In the Firefox dataset, prediction accuracy increases from 23.28% to 69.45%. Similarly, in the Eclipse dataset, accuracy rises from 70.01% to 89.28%. These results prove that including commit messages can significantly improve bug-triaging accuracy.**

***Index Terms*—Bug Triaging, Automation, Machine Learning, Natural Language Processing, Software Testing.**

## I. Introduction

Since bugs are a pervasive part of software development, fixing them is essential to preserving and improving the product's quality [1]. Frequently, bug reports are used to monitor the bug-resolution process since they provide thorough records of the problems found [2]. Effective bug-triaging and assigning these reports to the most suitable developer is crucial for quick and successful defect resolution throughout the software development lifecycle.

Triaging bugs takes a lot of effort and is difficult by nature. This duty has historically been overseen by senior team members and calls for long work hours. For example, Mozilla receives over 300 bug reports every day [3] and Eclipse reports around 37 per day [4]. The Mozilla code base has over 233 developers contributing to it [5]. It is difficult and error-prone to assign defects appropriately based on developer experience and bug requirements, which causes delays and frequent reassignment of reports. One way to address this is by automated bug triaging, which speeds up the assignment process while preserving human resources.

Recent work in automated bug triaging has integrated natural language processing (NLP) methods with machine learning (ML) models [3], [6], [7]. These techniques treat bug triaging as a problem of multi-class text classification. Conventional methods such as Word2Vec and TF-IDF (Term Frequency-Inverse Document Frequency) have been applied in multiple works [8], [9]. Although Word2Vec's static embedding is unable to adjust to changing contexts, reducing their expressiveness and accuracy when capturing intricate textual patterns, TF-IDF assesses term frequency but lacks semantic context.

Even with these developments, performance with existing Machine Learning (ML) models has frequently been less than ideal. Significant improvement has been demonstrated by transformer-based approaches, such as BERT (Bidirectional Encoder Representations from Transformers) [5], [10]. To capture complex word relationships and semantic context, these models make use of self-attention mechanisms [11]. Further improving this is XLNet (Generalised Autoregressive Pretraining for Language Understanding), which combines autoregressive and auto encoding techniques [12]. It does this by modeling complicated word dependencies and producing more accurate embedding by the use of permutation language modeling [13].

In this study, we propose a novel approach to enhance bug triaging by employing a feature engineering technique for improved dataset preparation and training. We use a state-of-the-art transformer-based neural network model to evaluate our methodology. Our results demonstrate a clear advantage over existing models i.e., showing substantial improvements in performance metrics [10], [12], [14]–[17]. Specifically, XLNet improved accuracy from 23.28% to 69.45% for the Firefox dataset and from 70.01% to 89.28% for the Eclipse dataset, underscoring its effectiveness to capture nuanced contexts of bug reports.

This paper is organized as follows: Section II reviews background and related work on bug triaging using various graph neural networks, machine learning, and NLP techniques. Section III describes our infrastructure setup for data collection, preprocessing, and research methodology. Section IV outlines our research prolongs and evaluation methods. Section V presents our results and highlights. Section VI concludes with a detailed discussion of our findings and explores potential future work to enhance the process.

## II. Background

Automated bug triaging is critical to addressing the inefficiencies of manual defect assignment as software systems

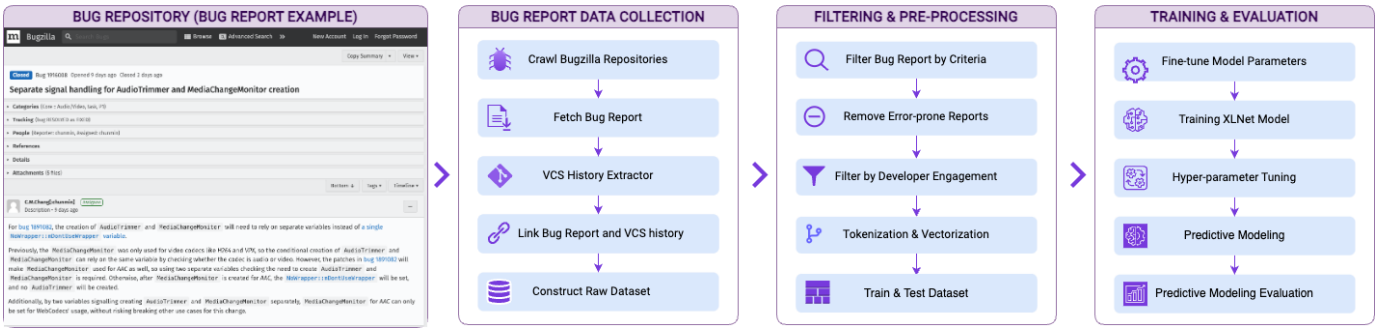


Fig. 1: An Overview of Data Collection, Training Transformer Model, and Evaluation

scale. Due to the growing number of bug reports, manual triaging is becoming increasingly problematic and prone to errors. Therefore, researchers are leveraging machine learning approaches to automate the triaging process to overcome these limitations. To address the issue in extensive open-source projects, this section examines the development of automated bug triaging, emphasizing advancements in ML and Natural Language Processing (NLP) that improve triaging efficiency and accuracy.

Conventional semi-automated methods for triaging bug reports use ML algorithms to examine past issue data and suggest developers who can address fresh reports. For instance, in the Eclipse and Firefox projects, prior techniques yielded precision levels of 57% and 64%, respectively [18]. Our method expands on this foundation by improving the dataset that models were trained on [19] to address the problem of ”tossing” bug reports—where reports are reallocated to other developers, lengthening the time-to-correction—this work uses a graph model based on Markov chains. This model effectively reduced tossing events by up to 72% and increased accuracy by up to 23%. The work of Kukkar et al. [20] focuses on automating developer recommendations for bug triage by creating a concept profile (CP) from bug reports and using social network (SN) data to identify suitable developers. While this method shows improved performance compared to other recommendation systems, however, due to limited contextual information and the narrow scope of Bug it has its limitations [5], [21]. The provided approach addresses bug assignments in large, long-lived software projects using a combination of ML techniques and probabilistic graph-based models. This approach aims to improve prediction accuracy and reduce software evolution costs by employing a diverse set of tools and evaluating their effectiveness on extensive datasets. However, the approach had a high dependence on Historical Data, the Complexity of the Implementation of the model as well as methodology Focused on Static Metrics. DevRec is another advanced study for recommending developers capable of resolving bugs [22], employing a dual-analysis framework that integrates bug report-based (BR-based) and developer-based (D-based) analyses. Although DevRec demonstrates superior performance compared to methods such as Bugzie [23] and DREX [24], particularly in terms of recall scores, it is not without limitations.

Although useful in many situations, the dual-analysis approach has certain built-in drawbacks. The BR-based analysis might not adequately capture the changing context of current difficulties, but it might offer insightful information based on previous bug reports. The D-based analysis can evaluate the skills and knowledge of developers, but it is not dynamic enough to account for changes in developer competence or recent modifications in the codebase, which can result in static developer affinity. As such, this may lead to a restricted contextual comprehension of the bug reports and may miss up-to-date, pertinent information, which could affect the accuracy of developer recommendations.

Applying machine learning techniques is the most popular approach in automated bug triaging. [3] first applied SVM (Support Vector Machine), a supervised machine learning algorithm that shows promises in automated bug triaging. Following the path, classical supervised ML algorithms such as Naive Bayes(NB) and Ensemble learning have been experimented with to produce better results [25]. Although traditional machine-learning approaches paved the way, the result was not satisfactory. With the evolution of Deep Learning (DL) [26], researchers started applying various DL methods like Convolutional Neural Networks(CNN) [27], Artificial Neural Networks(ANN) [28], and Deep bidirectional recurrent neural network with attention mechanism technique (DB-RNN-A) to advance the field. However, these techniques have limitations in grasping the contextual expressiveness of bug report features [29]. Studies with variants of BERT models that demonstrate unique abilities in terms of classification. Nevertheless, the bug report features used to train the BERT model were inadequate in achieving outstanding results.

Current automated approaches for triaging bugs usually concentrate on identifying the best developer by utilizing term selection strategies and allocating tasks according to developer experience. However, these methods have several limitations. For example, term selection methods often work on a restricted scope that can lead to the potentially omitted critical contextual information required for precise bug classification. The dynamic character of developer expertise and current

workloads may not be taken into consideration by experience-based load balancing, resulting in recommendations that are not ideal. Furthermore, insufficient context in problem reports often affects these methodologies, which might reduce the precision of triaging results.

## III. Data Collection and Preprocessing

We use the well-documented Eclipse [30] and Firefox [31] bug repositories in this project. These large datasets include a wide variety of structured modules, which makes them an excellent foundation for evaluating our methods in a variety of software settings and bug-reporting situations.

### A. Dataset Description and Characteristics

*1) Eclipse Dataset:* Eclipse is a feature-rich integrated development environment (IDE) with a broad range of plugins. We extracted 32,776 bug reports from four different modules - Core, Platform, UI, and IDE from the Eclipse Bugzilla repository.

*2) Firefox Dataset:* Firefox is a popular web browser with copious features. We gathered 9,999 bug reports. These two large-scale systems have a broad range of features and developers which facilitates a unique perspective for our methodological assessment.

### B. Data Preprocessing

Raw data often contains incomplete bug reports, missing metadata, or properties related to the bug that are not suitable for training models. We developed the following multi-step data filtering process to retain the quality and relevance of the dataset, as shown in Figure 1.

*1) Selection of Completed Bug Reports:* To ensure that the dataset includes only those reports for which the resolution and developer expertise are well-documented, we focused exclusively on bug reports marked as "FIXED."

*2) Exclusion of Incomplete Reports:* As missing or incomplete information causes noise and undermines the training process we excluded the reports lacking crucial metadata, such as title, description, or summary.

*3) Developer Engagement Filtering:* We removed bug reports from developers who fixed fewer than ten or more than fifty problems to mitigate potential biases from developers who were either too involved or too little involved. This method guarantees that the model stays impartial and does not give preference to developers who have a negligible or significant impact.

## IV. Experimental Framework, Algorithm Settings and Evaluation Metrics

### A. Experimental Framework

Our goal was to test how the inclusion of commit messages in bug reports affected the accuracy of developer assignments. We used well-documented datasets from Eclipse and Firefox bug repositories, which allowed us to evaluate our method across diverse software projects.

*1) Data Processing:* We utilized publicly available bug repositories from large-scale open-source projects to test the effectiveness of our model. After applying strict filtering criteria to ensure the data quality, we curved a subset of bug reports contain necessary metadata, including titles, descriptions and commit messages. The dataset was split into training, validations and test sets using a 80-10-10 ratio.

TABLE I: Dataset Description

| Project | Time Interval | #Bug Reports | #Developers |
|---|---|---|---|
| Firefox | July 13, 2004 - January 06, 2024 | 9,999 | 635 |
| Eclipse | October 19, 2001 - June 05, 2023 | 32,776 | 526 |

*2) Text Processing:* We preprocessed the text fields of the bug reports, like title, description, and commit messages to standardize our data. We converted all text to lowercase and removed punctuations and stopwords. Stemming is used to reduce words to their base forms, to reduce the dimensionality of the input data feature, and to focus on relevant features, as shown in Figure 1.

*3) Tokenization and Vectorization:* XLNet [12], a state-of-the-art transformer model, is particularly suited for contextual relationships within text sequences, which is imperative for understanding the nuances of bug reports and commit messages. We used the model to tokenize and vectorize the text data. Each text field was transformed into a sequence of numerical tokens for features of the model.

*4) Data Augmentation:* We leveraged the data augmentation techniques to prevent overfitting by exposing the model to a broader range of input scenarios during training. By introducing minor perturbations in the text, we generated synthetic variations of existing bug reports.

*5) Dataset Splitting:* Finally, we divided the dataset into validation (10%), training (80%), and test (10%). The test set assesses the model's ability to generalize to new, untested data, guaranteeing reliable performance evaluation; the validation set is utilized for hyperparameter tuning and model optimization; and the training set allows the model to learn from a variety of examples.

### B. Algorithm Settings

*1) XLNet Architecture:* We used a custom `XLNetForMultiLabelSequenceClassification` model. The architecture leverages the pre-trained `xlnet-base-cased` model. We have built our model with the consist of the following components:

- **XLNet Model**: We build our model from the pre-trained `xlnet-base-cased` model to extract contextual embedding from the input text sequences. The model captures bidirectional dependencies in the text using a permutation-based language model.
- **Classifier Layer**: Our model is a fully connected linear layer that is equal to the number of labels (developers) that were added on top of XLNet.
- **Initialization**: We used `Xavier initialization` for weight initialization of the classifier for faster conver-

TABLE II: XLNet Metrics in Eclipse and Firefox

| Features | Metrics | Eclipse (XLNet) | | | | Firefox (XLNet) | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Top-1 | Top-3 | Top-5 | Top-10 | Top-1 | Top-3 | Top-5 | Top-10 |
| Summary + Description | Accuracy | 47.89% | 64.44% | 70.01% | 77.26% | 23.28% | 40.78% | 48.36% | 62.89% |
| | F1 Score | 64.77% | 78.37% | 82.36% | 87.17% | 37.77% | 57.94% | 65.19% | 77.22% |
| Summary + Commit | Accuracy | 54.96% | 59.54% | 63.07% | 68.46% | 64.14% | 75.62% | 80.16% | 86.41% |
| | F1 Score | 70.93% | 74.64% | 77.36% | 81.28% | 78.15% | 86.12% | 88.99% | 92.71% |
| Summary + Description + Commit | Accuracy | **84.88%** | **88.29%** | **89.28%** | **90.89%** | **69.45%** | **81.48%** | **84.53%** | **89.14%** |
| | F1 Score | **91.82%** | **93.78%** | **94.34%** | **95.23%** | **81.97%** | **89.80%** | **91.62%** | **94.26%** |

gence by keeping the variance of the gradients consistent across layers.

- **Pooling Layer**: The hidden states produced by XLNet were pooled into a mean vector, reducing the sequence output to a single representation while retaining the most relevant information for classification.

*2) Freezing and Unfreezing XLNet:* During initial training, XLNet layers were frozen to prevent weight updates, allowing the classifier to train efficiently. Once stable, we unfroze the layers to allow fine-tuning, improving performance.

*3) Optimization Details:* We trained our XLNet-based model using the following optimization settings:

- **Optimizer**: We employed the `AdamW optimizer` for training, which is well-suited for transformer-based models. The optimizer updates the model weights while applying weight decay to prevent overfitting.
  - **Learning Rate**: The learning rate was set to $2e-5$, balancing rapid convergence and stable updates.
  - **Weight Decay**: We applied a weight decay of 0.01 to regularize the model.
- **Loss Function**: We used `Binary Cross-Entropy with Logits Loss (BCEWithLogitsLoss)` for our multi-label classification task. Thus, the loss function independently evaluates the probability of each developer being the correct assignee, treating each potential developer as a binary classification task.
- **Batch Size**: We have used a batch size of 16 to balance computational efficiency and learning performance.

*4) Training and Validation:* We trained the modelfor 20 epochs with an 80-20 training-validation split. Early stopping was applied to prevent overfitting, and the model was checkpointed based on validation loss. Training could be resumed from the latest checkpoint if interrupted.

### C. Evaluation Metrics

We are evaluating our model performance of Firefox and Eclipse Dataset with different features based on the below evaluation metrics.

*1. Acuracy*: As a percentage of all bug reports, accuracy measures how many bug reports are accurately classified. It quantifies the frequency with which the model accurately designates a bug to the relevant developer based on the input features, including commit messages, summary, and description of the bug report.

*2. F-1 Score*: The harmonic mean of precision and recall is computed to account for both erroneous positives and false negatives.

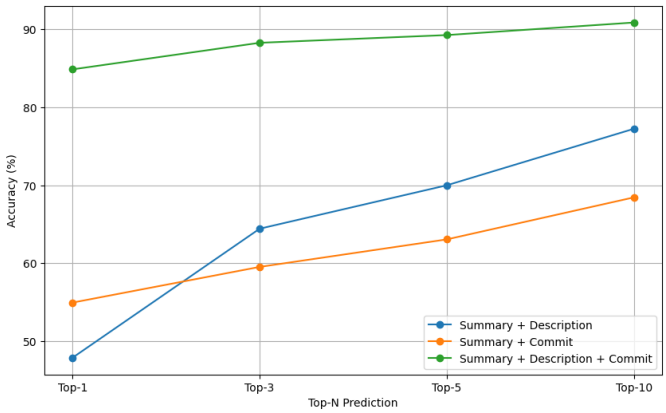


Fig. 2: Features Impact on Eclipse Triaging Result

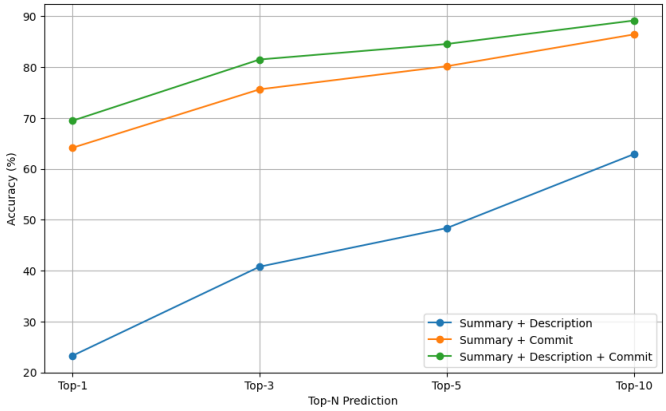


Fig. 3: Features Impact on Firefox Triaging Result

Our experiments are guided by three research questions.

**RQ1:** How Model Performs in terms of Accuracy and F-1 Score for Eclipse and Firefox Dataset Respectively when using features separately and combined without Commit Messages?

**RQ2**: How does the Model perform in terms of Accuracy and F-1 Score for Eclipse and Firefox Dataset, Respectively, when using features separately and combined, including Commit Messages?

**RQ3:** When using Firefox and Eclipse datasets, how does our model perform in terms of accuracy and F-1 score as compared to existing implemented models?

The study questions examine how different feature combinations impact a bug-triaging model's performance. We concentrate on F1 scores and Accuracy on the Eclipse and Firefox datasets. Performance in the first question is evaluated using

only the Summary + Description and Summary + Commit Messages from the bug report. The second question looks at the effects of using Summary + Description + Commit messages. The final question evaluates the performance of our transformer model with similar models implemented in the past.

TABLE III: XLNet vs. Transformer Models Accuracy on Eclipse and Firefox Datasets

| Dataset | Model | Top-1 | Top-5 | Top-10 |
|---|---|---|---|---|
| Eclipse | BERT | 29.8% | 66.3% | 79.1% |
| | ALBERT | 29.1% | 67.3% | 80.5% |
| | CodeBERT | 32.3% | 69.6% | 81.2% |
| | DeBERTa | 34.4% | 73.1% | 86.6% |
| | RoBERTa | 32.1% | 69.5% | 81.6% |
| | **XLNet** | **84.88%** | **89.24%** | **90.89%** |
| Firefox | BERT | 34.6% | 46.7% | 53.6% |
| | ALBERT | 39.2% | 50.9% | 45.7% |
| | CodeBERT | 33.1% | 44.1% | 50.6% |
| | DeBERTa | 65.3% | 79.5% | 83.8% |
| | RoBERTa | 35.4% | 46.5% | 52.5% |
| | **XLNet** | **69.45%** | **84.53%** | **89.14%** |

## V. Results and Discussion

We analyzed top-1, top-3, top-5, and top-10 prediction accuracies for each feature set across the developer pool to evaluate the model's predictive capability.

**RQ1**: Table II compares the performance of Summary + Description and Summary + Commit Messages in terms of F-1 Score and Accuracy for both datasets.

### *A. With Summary and Description*

Firefox: Top-1 accuracy is 23.28%, top-10 accuracy is 23.28%, and the top-10 F-1 Score is 77.22%. This performs better over a wider range of predictions despite weaker initial accuracy, as shown in Table II. Eclipse: Top-1 accuracy is 47.89%, top-10 accuracy is 77.26%, and the top-10 F-1 Score is 87.17%. The model performs better initially and across broader predictions.

### *B. With Summary and Commit Messages*

Firefox: Top-1 accuracy improves to 64.14%, top-10 accuracy is 86.41%, and the top-10 F-1 Score is 92.71%. Commit Messages to enhance precision and recall.

Eclipse: Top-1 accuracy is 54.96%, top-10 accuracy is 68.46%, and the top-10 F-1 Score is 81.28%. Commit messages provide an improvement, though less pronounced. Higher top-10 F-1 Scores reflect the model's improved ability to balance precision and recall across multiple predictions. The inclusion of Commit Messages significantly enhances performance, as shown in Figure 2 and 3.

**RQ2**: Table II shows the performance of all the combinations of features - Summary Description and Commit Messages. Seemingly, the result obtained incorporating the Commit messages outperforms the other.

Bugs are reported by end users and support engineers who don't have insights into the software systems but rather the features. Users' bug descriptions relate more to the features rather than the technicalities of the software system. On the contrary, the developers possess the ability to investigate the origin of the issue as a white-box view of the system. Traditionally, a developer commits an issue-fix with a proper message that expresses the root cause of the issue. Thus, commit messages with other bug report properties generate more likeliness to correlate a developer with a new bug report. Moreover, commit messages to boost the learning for the model when the texts of summary and descriptions are insufficient. Figure 2 and 3 depict the positive impact on predictive modeling when Commit messages are included in the feature set.

Figure 2 and 3 depict the positive impact on predictive modeling when Commit messages are added to the feature set. The best overall performance yields for Firefox's Top-1 accuracy is 69.45%, top-10 accuracy is 89.14%, and the top-10 F-1 Score is 94.26% and for Eclipse: Top-1 accuracy is 84.88%, top-10 accuracy is 90.89%, and the top-10 F-1 Score is 95.23%. The highest scores are achieved with all features combined, demonstrating significant improvements in all Top-N predictions.

**RQ3**: Our model outperforms current models, such as BERT, ALBERT, CodeBERT, DeBERTa, RoBERTa, and XLNet [10], [12], [14]–[17] for the Eclipse dataset across all accuracy measures (Top-1, Top-5, and Top-10), as shown in Table III. Similarly, our model outperforms other models significantly on the Firefox dataset, achieving the greatest accuracy across all criteria. Table III demonstrates our model's superior performance compared to existing models due to its integration of additional data sources, including system logs, developer history, and code snippets. This incorporation enhances the model's contextual understanding and significantly improves bug-triaging accuracy.

With the use of sophisticated contextual processing algorithms and sophisticated transfer learning architectures, our model grasps complicated language patterns with more accuracy and better F1 scores. These enhancements are validated by the empirical results from the Firefox and Eclipse datasets, which demonstrate our model's improved ability to efficiently and precisely categorize and prioritize issues.

## VI. Conclusion

In this project, we improved automated issue triaging by incorporating developer commit messages into our predictive models using the Eclipse and Firefox datasets. Utilizing the XLNet [12] model significantly enhances triaging performance. More specifically, for the Firefox dataset, top-1 prediction accuracy went from 23.28% to 69.45%, and for the Eclipse dataset, it went from 70.01% to 89.28%. This large uplift shows how effective it is to include commit messages—which offer important context—in the triaging process. More insights are added to defect reports by the comprehensive

commit data, which results in more precise developer assignments and fewer reassignment errors. Large-scale open-source projects like Mozilla Firefox and Eclipse, where the sheer number of bug reports and developers makes manual triaging difficult, benefit significantly from our method. Additionally, it can be optimized, and resolution times can shortened in various software development environments through automated triaging strategies.

## VII. Future Work

In the future, we aim to enhance our model to include data from other sources, such as system logs, developer histories, and code snippets. The subpar performance of summary and commit messages in the Eclipse dataset suggests that our first study may not be sufficient. Commit message opacity can negatively impact the quality of triaging, resulting in less accurate problem classification and prioritization.

We suggest that future studies focus on creating systems to extract and use high-quality commit messages. There is potential to improve significantly bug triaging systems by looking into ways to make commit messages more helpful and thorough. A more comprehensive grasp of the development context may be possible by including real-time data from code snippets, developer histories, and system logs into the triaging framework. This would further optimize the triaging procedure. Subsequent investigations can expand on our discoveries to create more resilient and flexible bug classification systems by tackling these aspects. Providing insightful information and valuable enhancements for bug-triaging techniques will contribute to the advancement of automated code analysis and software maintenance.

## VIII. Acknowledgments

This work was supported by a grant hidden for anonymous review.

## References

[1] S. Kim and E. J. Whitehead, "How long did it take to fix bugs?" in *Proceedings of the 2006 International Workshop on Mining Software Repositories*, ser. MSR '06. Association for Computing Machinery, 2006, p. 173–174.

[2] A. Arcuri, "On the automation of fixing software bugs," in *Companion of the 30th international conference on Software engineering*, 2008, pp. 1003–1006.

[3] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 361–370.

[4] J. Anvik, "Automating bug report assignment," in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 937–940.

[5] A. K. Dipongkor and K. Moran, "A comparative study of transformer-based neural text representation techniques on bug triaging," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 1012–1023.

[6] M. Alenezi, K. Magel, and S. Banitaan, "Efficient bug triaging using text mining." *J. Softw.*, vol. 8, no. 9, pp. 2185–2190, 2013.

[7] P. Bhattacharya, I. Neamtiu, and C. R. Shelton, "Automated, highly-accurate, bug assignment using machine learning and tossing graphs," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2275–2292, 2012.

[8] A. Aizawa, "An information-theoretic perspective of tf–idf measures," *Information Processing & Management*, vol. 39, no. 1, pp. 45–65, 2003.

[9] K. W. Church, "Word2vec," *Natural Language Engineering*, vol. 23, no. 1, pp. 155–162, 2017.

[10] J. Devlin, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[11] A. Vaswani, "Attention is all you need," *Advances in Neural Information Processing Systems*, 2017.

[12] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, and Q. V. Le, "Xlnet: Generalized autoregressive pretraining for language understanding," *Advances in neural information processing systems*, vol. 32, 2019.

[13] N. Arabadzhieva-Kalcheva and I. Kovachev, "Comparison of bert and xlnet accuracy with classical methods and algorithms in text classification," in *2021 International Conference on Biomedical Innovations and Applications (BIA)*, vol. 1. IEEE, 2022, pp. 74–76.

[14] Z. Lan, "Albert: A lite bert for self-supervised learning of language representations," *arXiv preprint arXiv:1909.11942*, 2019.

[15] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[16] P. He, X. Liu, J. Gao, and W. Chen, "Deberta: Decoding-enhanced bert with disentangled attention," *arXiv preprint arXiv:2006.03654*, 2020.

[17] K. L. Tan, C. P. Lee, K. S. M. Anbananthen, and K. M. Lim, "Roberta-lstm: a hybrid model for sentiment analysis with transformer and recurrent neural network," *IEEE Access*, vol. 10, pp. 21 517–21 525, 2022.

[18] S. Guo, X. Zhang, X. Yang, R. Chen, C. Guo, H. Li, and T. Li, "Developer activity motivated bug triaging: via convolutional neural network," *Neural Processing Letters*, vol. 51, no. 3, pp. 2589–2606, 2020.

[19] H. Dong, H. Ren, J. Shi, Y. Xie, and X. Hu, "Neighborhood contrastive learning-based graph neural network for bug triaging," *Science of Computer Programming*, no. C, p. 103093, 2024.

[20] A. Kukkar, U. K. Lilhore, J. Frnda, J. K. Sandhu, R. P. Das, N. Goyal, A. Kumar, K. Muduli, and F. Rezac, "Prore: An aco-based programmer recommendation model to precisely manage software bugs," *Journal of King Saud University-Computer and Information Sciences*, vol. 35, no. 1, pp. 483–498, 2023.

[21] S. F. A. Zaidi, F. M. Awan, M. Lee, H. Woo, and C.-G. Lee, "Applying convolutional neural networks with different word representation techniques to recommend bug fixers," *IEEE Access*, vol. 8, pp. 213 729–213 747, 2020.

[22] V. Dedík and B. Rossi, "Automated bug triaging in an industrial context," in *2016 42th Euromicro conference on software engineering and advanced applications (SEAA)*. IEEE, 2016, pp. 363–367.

[23] A. Tamrawi, T. T. Nguyen, J. Al-Kofahi, and T. N. Nguyen, "Fuzzy set-based automatic bug triaging (nier track)," in *Proceedings of the 33rd international conference on software engineering*, 2011, pp. 884–887.

[24] T. Zhang and B. Lee, "An automated bug triage approach: A concept profile and social network based developer recommendation," in *Intelligent Computing Technology: 8th International Conference, ICIC 2012, Huangshan, China, July 25-29, 2012. Proceedings 8*. Springer, 2012, pp. 505–512.

[25] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2009, pp. 111–120.

[26] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[27] K. O'Shea, "An introduction to convolutional neural networks," *arXiv preprint arXiv:1511.08458*, 2015.

[28] J. Zupan, "Introduction to artificial neural network (ann) methods: what they are and how to use them," *Acta Chimica Slovenica*, vol. 41, no. 3, p. 327, 1994.

[29] S.-R. Lee, M.-J. Heo, C.-G. Lee, M. Kim, and G. Jeong, "Applying deep learning based automatic bug triager to industrial projects," in *Proceedings of the 2017 11th Joint Meeting on foundations of software engineering*, 2017, pp. 926–931.

[30] Bug triage datasets: Eclipse ide, eclipse core, eclipse platform, eclipse ui by eclipse foundation. Accessed: Sept. 9, 2024. [Online]. Available: https://bugs.eclipse.org/bugs/

[31] Bug triage datasets: Mozilla firefox and mozilla core. Accessed: Sept. 9, 2024. [Online]. Available: https://bugzilla.mozilla.org/