

TriagerX: Dual Transformers for Bug Triaging Tasks with Content and Interaction Based Rankings

Md Afif Al Mamun¹, Gias Uddin², Lan Xia³, Longyu Zhang⁴

¹ University of Calgary, ² York University, ^{3,4} IBM Canada

Abstract—Pretrained Language Models or PLMs are transformer-based architectures that can be used in bug triaging tasks. PLMs can better capture token semantics than traditional Machine Learning (ML) models that rely on statistical features (e.g., TF-IDF, bag of words). However, PLMs may still attend to less relevant tokens in a bug report, which can impact their effectiveness. In addition, the model can be sub-optimal with its recommendations when the interaction history of developers around similar bugs is not taken into account. We designed TriagerX to address these limitations. First, to assess token semantics more reliably, we leverage a dual-transformer architecture. Unlike current state-of-the-art (SOTA) baselines that employ a single transformer architecture, TriagerX collects recommendations from two transformers with each offering recommendations via its last three layers. This setup generates a robust content-based ranking of candidate developers. TriagerX then refines this ranking by employing a novel interaction-based ranking methodology, which considers developers’ historical interactions with similar fixed bugs. Across five datasets, TriagerX surpasses all nine transformer-based methods, including SOTA baselines, often improving Top-1 and Top-3 developer recommendation accuracy by over 10%. We worked with our large industry partner to successfully deploy TriagerX in their development environment. The partner required both developer and component recommendations, with components acting as proxies for team assignments—particularly useful in cases of developer turnover or team changes. We trained TriagerX on the partner’s dataset for both tasks, and it outperformed SOTA baselines by up to 10% for component recommendations and 54% for developer recommendations. Replication package. <https://github.com/afifaniks/triagerX>

Index Terms—Bug Triage, Pre-trained Language Model, Ensembles, Text Embeddings, Developer Activity

I. INTRODUCTION

Bug triaging involves assigning reported issues to the most suitable developer or software team for resolution. Over the past few decades, various information retrieval (IR), machine learning (ML), and deep learning (DL) approaches automated this process [1]–[7]. However, their real-world adoption remains limited due to inconsistent performance across different datasets and industrial settings [8].

To understand and address these challenges, in collaboration with our industrial partner (IBM), we examined the limitations of existing approaches and then designed a novel bug triaging technique called TriagerX. We have successfully deployed TriagerX within the partner’s development environment.

TriagerX is built on the transformer architecture of Pre-trained Language Models (PLMs). This design decision is motivated by recent studies that demonstrated that PLMs outperform traditional ML and Information Retrieval (IR)

approaches in large-scale bug assignment tasks, benefiting from context-aware embeddings [5]–[7], [9]. Traditional TF-IDF-based methods rely on lexical similarity and are often less effective for natural language processing tasks where contextual meanings are important [10], [11].

Compared to the recent PLM-based bug triaging techniques, we innovate in TriagerX by offering a dual transformer architecture. This architecture decision is grounded on our studies on the effectiveness of current PLMs in the industrial setting. We fine-tuned top-performing PLMs identified in a recent comparative study [9] and compared them with TF-IDF, the state-of-the-art (SOTA) bug triaging model LBT-P (which uses a context-sensitive PLM) [7], and DBRNN-A [5], which relies on context-insensitive Word2Vec [12], using large-scale datasets from Mani et al. [5] as well as two newly prepared benchmarks: OpenJ9 and TypeScript (TS) (Section III-A1 discusses the datasets). Both TF-IDF and fine-tuned PLMs failed to deliver satisfactory results. Among prior works, the Knowledge Distillation (KD)-based [13] model LBT-P showed better performance on our partner’s datasets. However, LBT-P can be constrained by the teacher model’s generalization limits, affecting performance on small datasets like OpenJ9 [14]. Different PLMs tend to capture distinct characteristics, making them better suited to specific types of bug reports [9]. Moreover, PLMs are pretrained on general-domain corpora, and they often rely on shallow cues such as frequent phrasing, leading to reduced robustness on domain-specific tokens (e.g., software-related keywords) [15], [16]. Ensemble methods help mitigate these limitations by combining complementary model strengths [17], [18].

Based on these observations, we introduce a **Content-Based Ranker (CBR)** in TriagerX, a ranking model for bug triaging that combines embeddings from two PLMs and extracts features from multiple layers. By ensembling these embeddings, CBR leverages complementary information from both models, allowing it to focus on the orthogonality and diversity in the embeddings [19], [20]. The final recommendation is made by passing these combined representations through CNN-based classifiers. This approach improved accuracy while being smaller in size than large PLM-based models (see Section V-B). CBR outperformed baselines by up to 22% on literature datasets and 29% on partner datasets on Top-1 accuracy.

We also observed that human triagers in our partner teams prioritize developers who recently contributed to similar closed/fixed bugs through commits and comments. Frequent contributors are more likely to be assigned related bugs, yet recent PLM-based methods overlook this. While Yang et al.

previously proposed MDN [21] that exclusively used commit and comment counts for bug triaging, it performed poorly in our partner repositories due to its reliance on the smoothed Unigram Model (UM), which lacks contextual understanding, and its failure to prioritize recent contributions. To address this, we developed the **Interaction-Based Ranker (IBR)**, which enhances retrieval using a context-sensitive PLM and prioritizes developer contributions in a time-sensitive manner.

Listing 1. An example bug report from OpenJ9 (#19231)

Bug Title: Build-test doesn't support custom aqa-test repo.
Description: Launching a build via the Jenkins pipeline doesn't allow passing in a custom aqa-test repo. This complicates testing against excluded tests, requiring additional effort to ensure comprehensive test coverage without introducing regressions. FYI @llxia

In the OpenJ9 repository, IBR outperformed CBR by 4.5% (Top-1) and exceeded the best literature baseline by 34%. In TypeScript, IBR ranked second after CBR. This is due to repository dynamics—OpenJ9 has a stable maintainer group with frequent interactions, making past contributions more predictive. In contrast, TypeScript has a diverse, sporadic contributor base, reducing the impact of historical interactions.

Finally, we combined both CBR and IBR in **TriagerX** as a hybrid bug triaging framework that ranks developers using CBR and refines the ranking with IBR. To our knowledge, this is the first method to integrate interaction-based ranking with a PLM-driven approach. This version of TriagerX outperforms all methods, achieving Top-1 accuracy improvements of 54% and 26% over the best literature baselines, and 58% and 10% over the nearest large PLM models for OpenJ9 and TypeScript. For example, in Listing 1, despite @llxia being mentioned as the fixer, both LBT-P and standalone CBR in TriagerX misclassified the developer. However, TriagerX, combining CBR and IBR, correctly identified @llxia by considering interaction scores led to the correct prediction, as shown in Figure 1.

In summary, this paper makes the following contributions:

- **Dual-transformer content-based ranking.** We propose a novel Content-Based Ranker (CBR) that ensembles multiple pretrained language models and exploits multi-layer representations to capture complementary semantic signals from bug reports for triaging.
- **Interaction-aware ranking and hybrid integration.** We design an Interaction-Based Ranker (IBR) that models developers' historical activities through similarity-aware and time-decayed scoring, and integrate it with CBR via a unified hybrid ranking framework that balances textual relevance and developer interaction signals.
- **Real-world industrial deployment and validation.** We deploy TriagerX within IBM's OpenJ9 development workflow, demonstrating practical feasibility with low latency and strong real-world acceptance rates.

II. TRIAGERX: THE BUG TRIAGING FRAMEWORK

TriagerX has three major components: the Content-based Ranker (CBR), the Interaction-based Ranker (IBR), and the

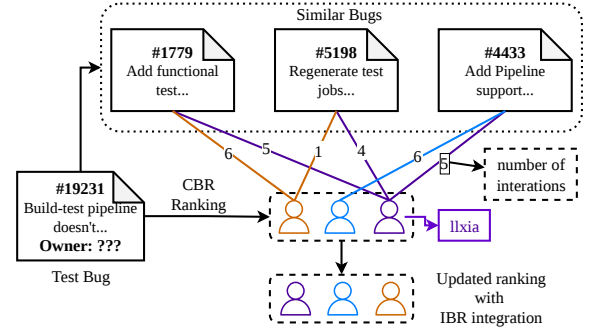


Fig. 1: TriagerX updated ranking with IBR for Listing 1.

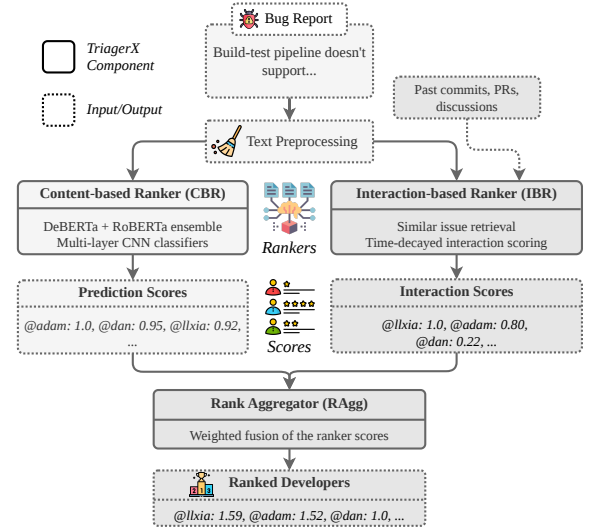


Fig. 2: Components and data flow of the TriagerX framework. A bug report (OpenJ9 #19231) is processed by CBR and IBR in parallel, producing prediction scores and interaction scores, respectively, which are fused by the Rank Aggregator (RAGg) to produce the final ranked developer list.

Rank Aggregator (RAGg). The CBR utilizes an ensemble of smaller PLMs to match the performance of larger models while reducing parameters. The IBR employs a scoring method based on developers' historical interactions (e.g., commits, pull requests, discussions/comments, etc.). Finally, the RAGg component applies an effective rank aggregation method to combine both CBR and IBR ranking scores, improving overall recommendation accuracy. Figure 2 illustrates the components and data flow of TriagerX using the bug report from Listing 1, showing how the Prediction Scores from CBR and the Interaction Scores from IBR are fused by the RAGg component to produce the final ranked list of developers.

A. Content-based Ranker

1) *Overall Architecture:* The Content-based Ranker illustrated in Figure 3 consists of two parts: PLMs and classifiers. Multiple PLMs extract rich semantic information from the textual representations of the bug data, providing diverse perspectives on the same bugs as discussed in Section I. Specifically, we select the base variants of DeBERTa and

RoBERTa for CBR due to their overall performance compared to the combination of other PLMs (Table VI).

We concatenate outputs from the last few hidden layers of 2 PLMs and pass them to classifiers, leveraging the hierarchical nature of PLM representations to capture various syntactic and semantic features. While the [CLS] token (a summarized/pooled representation of the input) from the final layer $H^{(L)}$ provides a context-rich representation, using multiple layers $H^{(L-K)}, \dots, H^{(L)}$ offers a broader feature set and is found to be more effective for downstream tasks [7], [10]. To account for varying contributions of hidden states, we use learnable weights $HW_{x,k}$ for each state, allowing adaptive weighting during classification. The weighted states from all PLMs are concatenated using Equation 1.

$$H_{concat} = \left(\left\|_{i=1}^x \text{PLM}_i(L-j) \times HW_{i,j} \right\|_{j=0}^{K-1} \right) \quad (1)$$

where L is the total number of layers, K is the number of classifiers, and x is the number of PLMs. Section IV-C discusses how using multiple PLMs enhances TriagerX CBR accuracy. Each classifier C_k processes its respective concatenated representation $H_{concat}^{(k)}$ to produce a prediction \hat{y}_k , which is weighted by learnable parameter CW_k that allows the model to adaptively learn and optimize classifier contributions during training for improved prediction. The combined prediction for CBR is calculated by Equation 2.

$$\hat{y}_{CBR} = \sum_{k=1}^K CW_k \times F(H_{concat}^{(k)}) \quad (2)$$

where K is the number of classifiers and $F(\cdot)$ is the CNN-based classifier function described in Section II-A3. This soft voting approach combines classifier outputs to make the final prediction, integrating diverse representations and improving model robustness [22], [23].

2) *PLM Fine-Tuning*: Fully fine-tuning a PLM can cause catastrophic forgetting [24], while freezing the PLM may lead to underfitting. Early stopping may prevent overfitting but limits learning potential. To address these challenges, we adopt Knowledge Preservation Fine-Tuning (KPFT) [7], which selectively freezes the initial layers to retain general language understanding and adapts the later layers to task-specific data. This balances the retention of prior knowledge with the flexibility to learn new representations. In our setup, for a PLM with L total layers, during KPFT we define the training status of each layer θ_l as:

$$\theta_l = \begin{cases} \text{trainable,} & \text{if } l \geq L - K + 1 \\ \text{frozen,} & \text{if } l < L - K + 1 \end{cases} \quad (3)$$

where K denotes the number of classifiers being trained. By freezing the majority of the PLM's encoder layers and selectively updating only those necessary for the downstream task, KPFT ensures that each classifier can effectively utilize task-specific representations while preserving the broader linguistic knowledge encoded in the initial layers.

3) *Developer Classifier*: Fully connected networks (FCNs) are generally used on PLM embeddings. However, our experiments show that simple FCNs may not provide optimal results for bug-triaging task (Section IV-D). To improve classification

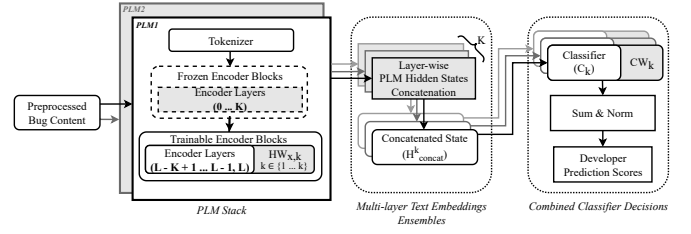


Fig. 3: CBR with 2 embedding modules and 3 classifiers.

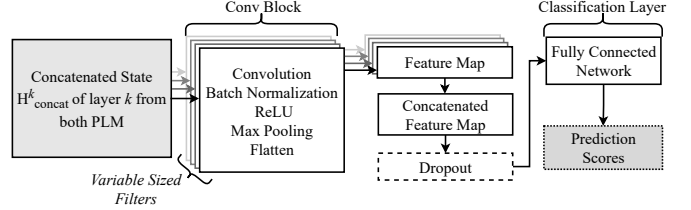


Fig. 4: Classifier architecture of TriagerX CBR.

accuracy, we adopt a CNN-based architecture, visualized in Figure 4. CNNs capture local patterns and hierarchical features more efficiently, making them suitable for processing concatenated hidden states from the PLMs. Following prior work, each convolution block (CB) uses four parallel convolution layers with filter sizes $f_s \in \{3, 4, 5, 6\}$, each with 256 filters allowing the model to capture features of different n-gram lengths [7]. Each convolution layer is followed by batch normalization (BN) [25], ReLU activation, max pooling and flattening. Finally, we concatenate the feature maps from the parallel convolution layers. Before passing the feature vector to the FCN for the recommendation task, we apply a dropout layer [26] to minimize overfitting by randomly setting a fraction of inputs to zero during training.

B. Interaction-based Ranker

We introduce a novel interaction scoring technique to evaluate developers based on past involvement with similar bugs. Each developer receives an interaction score (IS) from their historical contributions, which adjusts the prediction scores from CBR to improve accuracy. The IBR technique begins by identifying similar issues using Siamese BERT networks (SBERT) [27]. We leverage a pre-trained SBERT model from the official repository as these models provide strong text similarity capabilities off-the-shelf without requiring additional fine-tuning. SBERT efficiently encodes sentences into dense vector representations, allowing fast and accurate retrieval of similar issues via cosine similarity, as defined in Equation 4.

$$\text{Similarity}(i, j) = \frac{\mathbf{e}_i \cdot \mathbf{e}_j}{\|\mathbf{e}_i\| \|\mathbf{e}_j\|} \quad (4)$$

where \mathbf{e}_i and \mathbf{e}_j are the dense vector embeddings of issues i and j , respectively, and issues i and j are considered similar if their similarity meets the threshold τ :

$$\text{IsSimilar}(i, j) = \begin{cases} \text{similar,} & \text{if } \text{Similarity}(i, j) \geq \tau \\ \text{dissimilar,} & \text{if } \text{Similarity}(i, j) < \tau \end{cases} \quad (5)$$

Since SBERT models generate context-sensitive embeddings independent of term frequencies between reports unlike [21], embeddings for all existing issues can be precomputed and stored, enabling real-time and accurate retrieval of similar issues. Once retrieved, timeline events of each similar issue are traversed to identify developers who interacted with it through commits, pull requests, discussions, or assignments. Each interaction is scored differently based on its type. For example, a commit to an issue may signify a developer’s involvement more than a comment. Contributions from *inactive developers* are excluded by maintaining a directory of *active developers* (D). We define an active developer as an individual who contributes a minimum number of times to a repository.

To account for temporal relevance, we weigh developer interactions using an exponential decay function $f(t) = e^{-\lambda t}$, where λ is the decay rate and t is the interval between the interaction date and the report time of the new issue being triaged, so that only interactions from issues reported before the new issue are considered. This approach ensures that recent contributions have a stronger influence on the ranking, while older interactions are smoothly and progressively discounted. Intuitively, a developer who interacted with a similar issue just two days ago is more likely to work on a related issue than someone who last interacted a year ago, even if both were involved in similar components. We adopt exponential decay over linear or stepwise alternatives because it offers a smooth, continuous discounting that aligns with how developer engagement typically fades over time. Prior work in collaborative filtering also showed that exponential decay more effectively models evolving user preferences and time-aware relevance [28], [29]. The total interaction score (IS) of a developer d_i for a new issue (I_{new}) is calculated by Equation 6.

$$IS(d_i) = \sum_{j=1}^m \sum_{k=1}^{n_{ij}} \text{Sim}(I_{new}, I_j) \cdot IP_{ijk} \cdot e^{-\lambda t_{ijk}} \quad (6)$$

where m is the number of similar issues retrieved for the new issue (I_{new}), n_{ij} is the total number of interactions by developer d_i on issue I_j , while $\text{Sim}(I_{new}, I_j)$ is the cosine similarity between I_{new} and retrieved issue I_j (Equation 4) weighting each interaction by the relevance of the issue it occurred on. IP_{ijk} is the interaction point value for the k -th interaction of developer d_i on issue I_j , drawn from a predefined table based on interaction type (e.g., commit/PR, discussion, assignment — see Table II), λ is the time decay rate controlling how quickly older interactions are discounted, and t_{ijk} is the elapsed time in days between the k -th interaction on issue I_j and the report time of the new issue I_{new} . Once scoring is done for all developers, scores are normalized by Equation 7.

$$NIS(d_i) = \frac{IS(d_i) - \min(IS)}{\max(IS) - \min(IS)} \quad (7)$$

Normalization is only done if there is at least one developer with a non-zero interaction score to avoid division by zero. The overall algorithm is provided in Algorithm 1.

Algorithm 1: IBR developer scoring algorithm based on previous interactions.

Input : New Issue I_{new} , Similarity threshold τ , Time decay factor λ , Existing all issues I_{all} , Existing issues embedding $\mathbf{E} = \{\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_N\}$, Interaction point table IT , Active developers $D = \{d_1, d_2, \dots, d_n\}$

Output: Normalized interaction scores (NIS)

- 1 Compute new issue embedding $\mathbf{e}_{new} = \text{SBERT}(I_{new})$;
- 2 Calculate cosine similarities with existing issues:
 $\text{Sim}(I_{new}, I_j) = \frac{\mathbf{e}_{new} \cdot \mathbf{e}_j}{\|\mathbf{e}_{new}\| \|\mathbf{e}_j\|}, \forall I_j \in I_{all}$;
- 3 Select top similar issues
 $I_{top} = \{I_j \mid \text{Sim}(I_{new}, I_j) \geq \tau\}, \forall I_j \in I_{all}$;
- 4 Initialize a score dictionary $IS(d_i) = 0, \forall d_i \in D$;
- 5 **foreach** issue $I_j \in I_{top}$ **do**
- 6 **foreach** interaction k by developer d_i on issue I_j **do**
- 7 **if** $d_i \in D$ **then**
- 8 Get interaction point IP_{ijk} from IT ;
- 9 Update $IS(d_i) =$
 $IS(d_i) + \text{Sim}(I_{new}, I_j) \cdot IP_{ijk} \cdot e^{-\lambda t_{ijk}}$;
- 10 **end**
- 6 **end**
- 7 **end**
- 8 **if** $\exists d_i \in D$ such that $IS(d_i) > 0$ **then**
- 9 Normalize Interaction Scores (NIS) using Equation 7
- 10 **return** NIS;
- 8 **end**
- 9 **return** IS;

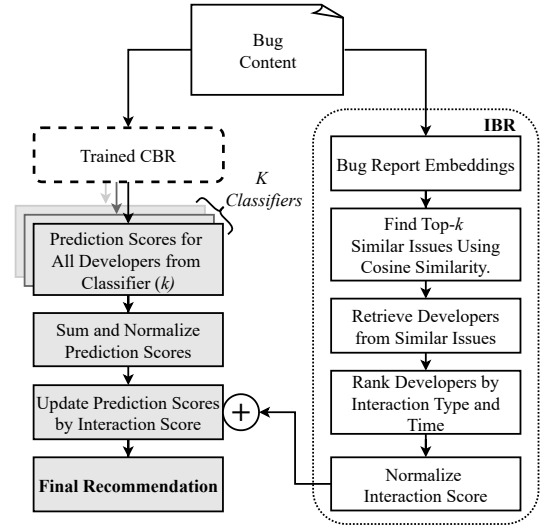


Fig. 5: TriagerX complete framework with both CBR and IBR.

C. Rank Aggregator

We employ a Weighted Ranking Aggregation (WRA) method that combines the normalized prediction score (NPS) with the weighted normalized interaction score (NIS). Developers who have not interacted to any similar issues retain their original prediction score from TriagerX CBR. The interaction scores are multiplied by a weighting factor W_f , which is optimized through grid search within the range $(0, 1)$ to ensure appropriate weighting of the interaction score depending on the dataset/repository. The calculation of the final score (FS)

for developer d_i is expressed as:

$$FS(d_i) = NPS(d_i) + W_f \cdot NIS(d_i) \quad (8)$$

Figure 5 shows the complete architecture of the TriagerX framework, including both CBR and IBR.

III. EVALUATION

Our goal is to study whether and how TriagerX outperforms the SOTA tools. We answer five research questions (RQs):

- RQ1.** Can TriagerX outperform the baselines?
- RQ2.** How effective are the ranker components individually?
- RQ3.** How do multiple PLMs influence the accuracy of CBR?
- RQ4.** Do the design decisions impact CBR performance?
- RQ5.** Do hyperparameter choices affect IBR performance?

RQ1 and RQ2 evaluate the framework’s accuracy and check the statistical significance of the results, with RQ1 focusing on overall accuracy and RQ2 assessing individual ranker components. RQ3 examines the effectiveness of combining multiple PLMs. RQ4 and RQ5 explore different design choices in the overall framework in the form of an ablation study.

A. Experimental Setup

1) *Benchmark Datasets:* We utilized large-scale Google Chromium (GC), Mozilla Core (MC), and Mozilla Firefox (MF) datasets from Mani et al. [5]. These datasets include issue title, description, reporting date, and the owner who resolved the bug. As the literature datasets do not contain developer interaction information, we also prepared two new benchmark datasets for our analysis. To create our own benchmarks, we leveraged the GitHub API to collect data from the OpenJ9 and TypeScript (TS) bug repositories. We gathered all reported bugs up to June 2024, dating back to the creation of each repository. These datasets include information such as issue titles, descriptions, assigned developers, and contributors (e.g., those who commented, committed, or created pull requests). Directly assigned developer to an issue is considered the owner of a bug report. In cases where there is no direct assignment on the GitHub issue page, we considered the last person to make a commit or pull request to that bug as the owner. If neither of this information was found on the issue page, we discarded the issue from our dataset.

Text Preprocessing. We applied consistent preprocessing across all datasets. First, we concatenated the issue title and description, then filtered out reports with fewer than 15 words to ensure sufficient context. URLs, timestamps, special characters, and code block identifiers (e.g., ```) were removed. Hexadecimal values were replaced with `<hex>` tokens to maintain log structure without introducing noise. For example, "Memory error at 0x7fff5fbff7a0" was transformed to "Memory error at `<hex>`." We used dynamic tokenization to ensure each PLM uses its respective pretrained tokenizer. Texts were tokenized to a maximum of 256 tokens, and shorter texts were padded.

Sampling Active Developers. In most software repositories, a small subset of developers contribute frequently, while the majority have limited, sporadic contributions—especially in open-source projects. To improve ML model generalization,

TABLE I: Datasets after preprocessing. Mean[†] and Std[†] refer to the average and standard deviation of issues owned per active developer.

Dataset	Time Range	Train	Test	Devs	Mean [†]	Std [†]
GC	08/2008 – 07/2016	98,165	10,781	1,032	110.5	115.4
MC	04/1998 – 06/2016	108,607	10,994	557	214.7	342.3
MF	07/1999 – 06/2016	18,223	1,755	155	128.9	167.6
OpenJ9	09/2017 – 06/2024	3,375	382	50	74.0	76.4
TS	07/2014 – 06/2024	9,929	1,129	40	276.2	353.6

often, only frequent contributors, termed *active developers* [2], [30], [31], are considered. We define an active developer D as one with contributions $C(D) \geq 20$, excluding those with $C(D) < 20$ to reduce outliers from insufficient training data. The threshold of 20 is motivated by Mani et al. [5], who observed that deep learning models show improved accuracy as the threshold increases up to 20, after which gains plateau. This filtering accounts for the difference in the number of issues and active developers shown in Table I after preprocessing, compared to one of our baselines, LBT-P [7].

Dataset Characteristics. We analyze the characteristics of all datasets after preprocessing, with key statistics summarized in Table I. Mozilla Core shows the highest variability (mean: 214.7, std: 342.3), driven by a small number of extremely active developers, while OpenJ9 is the most balanced (mean: 74.0, std: 76.4). TypeScript shows a similar degree of skewness to Mozilla Core (mean: 276.2, std: 353.6) despite having only 40 active developers, with the top 5 developers alone accounting for 48.3% of all resolved issues. We additionally analyze label diversity for our benchmark datasets (TypeScript and OpenJ9) as the literature datasets sourced from Mani et al. [5] do not include label or interaction metadata beyond issue titles, descriptions, and developer assignments. TypeScript issues carry an average of 2.27 labels per issue across 127 unique labels (e.g., *Bug*, *Suggestion*), reflecting a broad and diverse issue taxonomy. OpenJ9 issues carry an average of 2.17 labels across 84 unique labels, dominated more by component-specific labels such as *comp:vm* and *comp:jit*, reflecting a more structured labeling scheme consistent with OpenJ9’s stable maintainer group.

2) *Content-based Ranker Training:* We trained the CBR model across all datasets, using three classifier blocks on the concatenated hidden states from the final three encoder layers of the PLMs. Following literature baselines [5], [7], datasets were split chronologically, with 90% for training and 10% for testing. Although we focus on active developers, contribution levels vary significantly. For instance, in the GC dataset, the top contributor has over 1100 contributions, while the lowest after filtering has 20, creating class imbalance. To address this, we used weighted random sampling, assigning each issue a weight inversely proportional to its developer’s frequency:

$$P_i = \frac{1/f_i}{\sum_{j=1}^n (1/f_j)} \quad (9)$$

where P_i is the probability of selecting issue i , f_i is the frequency of issues assigned to developer d_i , and n is the total number of issues. This approach balances the training data

by sampling issues associated with less frequent developers more often. We used the AdamW optimizer [32] with a weight decay of 0.001 and a linear learning rate schedule, including a 1×10^{-5} peak learning rate and 10% warmup. Models were trained for up to 40 epochs on an NVIDIA A100 GPU.

3) *Interaction-based Ranker Optimization*: TriagerX IBR uses the pre-trained SBERT model *all-mpnet-base-v2* [33] for embedding generation without requiring task-specific fine-tuning, chosen for its superior accuracy over other models. Detailed comparisons between these models are not provided as they are not the focus of our research. For the experimental setup, embeddings for all training issues are pre-computed. When a new bug is reported, similar bugs from the existing issues are identified based on embedding similarity. Interaction scores are subsequently calculated by considering the nature of interactions with similar issues, including both interaction type and timing.

Optimization Process. To adapt interaction scoring to different repository settings, IBR avoids relying on fixed heuristic values and instead performs a straightforward hyperparameter tuning step to adjust key parameters (e.g., λ , IP_x) for optimal performance. Given the small number of parameters and a constrained search space, we employ an exhaustive grid search over all possible combinations of parameter values. Optimization is performed using a validation set sampled from less than 10% of the training data. Let $f(\mathbf{P})$ represent the accuracy of the framework evaluated at a specific parameter set \mathbf{P} within the defined grid of parameters $\mathcal{P} = \{(\tau, \lambda, IP_x, W_f) \mid \tau \in [\tau_{\min}, \tau_{\max}], \lambda \in [\lambda_{\min}, \lambda_{\max}] \dots\}$. The optimal parameter set is determined as follows:

$$\mathbf{P}_{\text{best}} = \arg \max_{\mathbf{P} \in \mathcal{P}} f(\mathbf{P}), \quad (10)$$

where \mathbf{P}_{best} is the parameter set that maximizes the accuracy $f(\mathbf{P})$ across the grid \mathcal{P} . Each dataset may require specific parameter adjustments due to unique characteristics like bug type distributions and developer activity patterns. We discuss these parameters and their search space below:

Similarity Threshold $\tau \in [0.2, 0.8]$. Filters semantically similar issues based on cosine similarity. Scores below 0.2 are typically unrelated and thus excluded. Conversely, issues with similarity above 0.8 are highly likely to be relevant. Hence, we limit the upper bound of the search range to 0.8 to reduce the search space. To ensure computational efficiency, similar issue retrieval is capped at a maximum of 20 issues per query. This prevents unbounded costs in both retrieval and subsequent interaction score calculation from bug reports at low threshold values explored during grid search or in deployment to repositories with large numbers of existing issues.

Time Decay Factor $\lambda \in [0.001, 0.01]$. This parameter controls how quickly interaction scores decay over time. Similar to prior research [28], we allow this wide range to adapt to different repository dynamics flexibly.

Interaction Points $IP_x \in [0, 2]$. Allows the framework to learn the relative importance of different developer interactions—assignments, commits/PRs, and discussions. The range is designed to represent three interpretable levels (0: low, 1:

TABLE II: Optimized hyperparameters for interaction-based ranker on different datasets obtained via grid search.

Parameter	Openj9	TS	Search Space	Step
Similarity Threshold τ	0.4	0.45	[0.2, 0.8]	0.05
Time Decay Factor λ	0.01	0.001	[0.001, 0.01]	0.001
Interaction Point (Assignee) IP_a	0.5	0.5	[0, 2]	0.10
Interaction Point (Commit/PR) IP_c	1.5	1.5	[0, 2]	0.10
Interaction Point (Discussion) IP_d	0.20	0.10	[0, 2]	0.10
Interaction Score Weight W_f	0.65	0.25	(0, 1)	0.05

medium, 2: high), with fractional values (e.g., 0.5, 1.5) enabling finer control. As shown in Table II, grid search typically assigns higher weights to commits/PRs than to discussions, reflecting their greater predictive value in developer activity.

Interaction Score Weight $W_f \in (0, 1)$. W_f controls the contribution of IBR relative to CBR during final rank aggregation (Equation 8). This parameter allows the framework to adjust the weight of IBR ranking scores. For example, a lower weight is found to be more effective in repositories with sparse interactions (e.g., TS), while higher weights benefit repositories with dense interaction histories (e.g., OpenJ9).

Table II presents the optimized hyperparameters for the OpenJ9 and TS datasets, along with their corresponding search ranges and step size (i.e., interval between consecutive values explored) used during the tuning process.

B. SOTA Baselines

We used the following approaches as our baselines.

LBT-P. is a bug triage framework using patient knowledge distillation (PKD) approach to compress RoBERTa-large that attempts to mitigate PLM’s overthinking problem [34]. As the official source code was not publicly available, we contacted the authors and were informed that the implementation is proprietary. Nevertheless, they shared partial code snippets and detailed reproduction guidelines. Following the paper and the authors’ instructions, we reimplemented the framework. On the GC dataset, our reproduction achieved results comparable to those reported in the original study, providing confidence in the correctness of our implementation.

DBRNN-A. is a deep bidirectional RNN with Attention that uses LSTM units to capture context in bug reports, addressing the challenge of mixed text, code snippets, and stack traces. Our reproduction of DBRNN-A following this implementation yields similar results to the original paper.

MDN. Multiple Developer Network first attempts to find similar issues by applying smoothed Unigram Model (UM) and Kullback-Leibler (KL) divergence. Then it generates a network of developers by the number of commits and comments on those bug reports.

Large PLMs with FCN and CNN classifiers. We evaluated high-performing PLM variants for bug triaging, as reported in [9]. All PLMs were sourced from the HuggingFace repository [35] and served as the core text embedding modules. We used the large variants of BERT [10], RoBERTa [36], and DeBERTa [37], given their typical superiority over base versions, along with the base variant of CodeBERT [38], as it is the only publicly available option. For FCN-based classifiers, we used the final-layer [CLS] token representation as the sequence-level embedding. For CNN-based classifiers, we applied con-

volutional filters (same CNN configuration as CBR without ensembling) over the hidden states from the last transformer layer, followed by pooling to obtain a fixed-size representation. **TF-IDF+SVM.** We use term frequency–inverse document frequency (TF-IDF) vectors to represent bug report text, capturing word-level importance across the corpus. These features are then fed into a linear Support Vector Machine (SVM) classifier, serving as a non-neural baseline for bug triaging.

We also evaluated recent graph-based bug triaging methods, including NCGBT [39] and PCG [40], using their official implementations. Both approaches performed poorly on our datasets. NCGBT relies on CC fields that are absent in our data, which likely explains its sharp performance drop as the dataset size increases, achieving only 0.035 Top-1 accuracy on OpenJ9 and ≈ 0.004 on Google Chromium. Similarly, while PCG is not dependent on CC data, it exhibited very low Top-k accuracy on large projects, consistent with the original paper’s report for Google Chromium (Top-1: 0.008) [40]. For completeness and transparency, we include these results in Appendix B and provide reproduction artifacts and hyperparameter configurations for these methods in our replication package, but omit them from the main paper due to their inferior performance on our datasets compared to many other baselines.

C. Evaluation Metric

Following our literature baselines [5], [7], we employ the Top-k metric to evaluate performance. Multiple developers may be qualified to fix a bug. Therefore, Top-k accuracy is defined as:

$$\text{Top-k Accuracy} = \frac{1}{N} \sum_{i=1}^N \mathbb{I}(y_i \in F(x_i, k)) \quad (11)$$

N is the number of bugs, y_i is the actual developer assigned to bug i , x_i is the bug’s input, $F(x_i, k)$ is the set of Top-k predicted developers, and \mathbb{I} is the indicator function that returns 1 if y_i is among the Top-k predictions, and 0 otherwise. We apply the Wilcoxon signed-rank test against the best-performing baseline in Top-1 position to check the significance in the performance difference ($p < 0.01$).

IV. RESULTS

A. RQ1: Can TriagerX outperform the baselines?

Approach. Evaluating our end-to-end framework on literature datasets is challenging due to the lack of detailed interaction data (assignments, commits, pull requests, temporal data, etc.). Hence, we created our own dataset. We compare the Top-k accuracy against the baselines.

Results. Table III shows that our framework consistently outperforms all baselines across Top-k metrics for $k \in \{1, 3, 5, 10, 20\}$ with statistical significance except for the Top-20 position in the TS dataset. On the OpenJ9 dataset, TriagerX achieves 54% higher Top-1 accuracy than LBT-P and 58% higher than the closest large PLM-based baseline, RoBERTa-large with CNN classifiers. CodeBERT showed the lowest performance across both datasets, while TF-IDF-based

SVM achieved competitive results, just behind LBT-P in Top-1 accuracy on the TS dataset and ranking just behind TriagerX and RoBERTa-based models. However, TF-IDF’s Top-k accuracy declines drastically for $k > 1$ compared to PLM-based approaches. Overall, TriagerX consistently outperformed all models on both datasets. Optimizing hyperparameters for the framework for smaller k values, such as Top-1 or 3, using our grid search technique (see Section III-A3) leads to improvements for larger k values as well. Despite TriagerX’s superior performance compared to other approaches, it still has a low Top-1 accuracy. To understand the misclassifications, we examined the examples and identified dataset imbalance as a key issue, with some developers having far more contributions than others. This bias leads the model to favor more active developers, resulting in poorer performance for those with fewer contributions. Another common reason is when the IBR interaction score fails to complement the misclassification by CBR. For instance, in OpenJ9 issue #17738, @JasonFengJ9 is the actual owner, but CBR predicted other developers who work on similar issues. Manual investigation revealed that similar issues used to calculate interaction scores had few interactions from @JasonFengJ9, thus failing to correct the final recommendation.

RQ1 Summary: The TriagerX framework outperforms the LBT-P baseline by up to 54% in Top-1 accuracy and also surpasses larger PLM-based approaches consistently.

B. RQ2: How effective are each of the ranker components?

Approach. We compare the Top-k accuracy of TriagerX IBR and CBR, evaluated separately.

Effectiveness of TriagerX IBR. We evaluate TriagerX IBR on partner datasets, as literature datasets lack developer interaction data. Table III shows that IBR, using similarity-based retrieval, outperforms other methods in Top- $\{1,3\}$ accuracy on the Openj9 dataset and performs competitively on the TS dataset signifying the effectiveness of using developer interaction data for bug triaging. In contrast, MDN underperforms on both datasets due to its reliance on unigram models that focus on term frequency rather than contextual meaning. For example, MDN may link a “VM crash” issue to a “UI crash” rather than a more relevant “JVM failure.” Additionally, the approach weighs developer contributions equally regardless of time which may prioritize developers who frequently contributed to similar issues long ago than someone who recently contributed to similar issues. Our approach overcomes these limitations by using a context-sensitive PLM and time-based contribution scoring. However, integrating both IBR and CBR delivers the best overall performance across both datasets. We analyzed misclassifications for IBR and found that team leads or project managers often engage with issues through discussions or mentions without being the actual solvers. Despite assigning low scores to discussions, these interactions can still lead to inaccurate recommendations. For instance, in OpenJ9 Issue #19197, user @pshipton frequently participated in discussion but was not the fixer. The IBR algorithm might incorrectly identify them as a potential resolver for similar issues, slightly lowering Top-1 accuracy.

TABLE III: Top-k accuracy of TriagerX framework compared to all baselines. P-values are calculated against the best-performing baseline (♣) where “*” indicates $p < 0.01$.

Dataset	Method	Top-1	Top-3	Top-5	Top-10	Top-20
Openj9	TriagerX (CBR+IBR)	0.327*	0.533*	0.633*	0.807*	0.918*
	TriagerX CBR	0.272	0.476	0.601	0.780	0.901
	TriagerX IBR	0.284	0.488	0.585	0.699	0.860
	DeBERTa-Large (FCN)	0.178	0.418	0.547	0.698	0.897
	RoBERTa-Large (FCN)	0.191	0.418	0.586	0.743	0.890
	BERT-Large (FCN)	0.168	0.393	0.507	0.694	0.857
	CodeBERT (FCN)	0.129	0.331	0.476	0.689	0.849
	DeBERTa-Large (CNN)	0.170	0.374	0.503	0.675	0.853
	RoBERTa-Large (CNN)	0.206	0.403	0.531	0.670	0.822
	BERT-Large (CNN)	0.181	0.323	0.445	0.652	0.839
	CodeBERT (CNN)	0.100	0.253	0.409	0.595	0.805
	LBT-P ♣	0.211	0.407	0.501	0.631	0.797
	DBRNN-A	0.127	0.300	0.454	0.627	0.775
	MDN	0.100	0.349	0.422	0.606	0.746
TF-IDF + SVM	0.189	0.357	0.484	0.665	0.828	
P-value	1.4e-08	3.4e-05	1.8e-07	1.8e-09	3.0e-13	
TS	TriagerX (CBR+IBR)	0.353*	0.615*	0.711*	0.830*	0.930
	TriagerX CBR	0.324	0.582	0.682	0.812	0.920
	TriagerX IBR	0.278	0.487	0.564	0.650	0.720
	DeBERTa-Large (FCN)	0.264	0.509	0.618	0.794	0.924
	RoBERTa-Large (FCN) ♣	0.319	0.552	0.669	0.824	0.929
	BERT-Large (FCN)	0.253	0.481	0.614	0.784	0.906
	CodeBERT (FCN)	0.120	0.309	0.458	0.733	0.915
	DeBERTa-Large (CNN)	0.212	0.428	0.580	0.765	0.918
	RoBERTa-Large (CNN)	0.294	0.495	0.602	0.739	0.876
	BERT-Large (CNN)	0.151	0.345	0.502	0.705	0.893
	CodeBERT (CNN)	0.143	0.352	0.506	0.704	0.890
	LBT-P	0.279	0.503	0.627	0.781	0.908
	DBRNN-A	0.231	0.447	0.579	0.729	0.838
	MDN	0.075	0.100	0.275	0.475	0.525
TF-IDF + SVM	0.272	0.428	0.493	0.663	0.830	
P-value	8.7e-11	1.4e-04	4.2e-03	4.4e-03	1.2e-01	

Effectiveness of TriagerX CBR. Since TriagerX CBR relies solely on the textual content of bug reports, we had a broader range of evaluation options. Table III presents the evaluation results of CBR for our datasets, and Table IV provides a comparative analysis with literature datasets. CBR consistently outperforms the literature baselines like LBT-P and DBRNN-A. CBR also achieves higher Top-k accuracy compared to larger PLMs with naive-FCN or CNN classifiers. DeBERTa performed better in the GC dataset, while RoBERTa in the MC dataset. However, TF-IDF performed significantly worse on larger datasets like GC and MC, likely due to its reliance on sparse feature representations. We also analyzed cases where the CBR model failed to predict the correct developer, often due to data imbalance. Additionally, the model sometimes makes incorrect recommendations when multiple developers work on similar issues. Although different developers might resolve the issue, these predictions are considered incorrect because the datasets assign each bug to only one developer. For example, consider the following Openj9 issues #18871 and #18873 that are extremely similar in title and with the same component labels but were fixed by different developers.

Analysis of IBR Contributions. We further analyze OpenJ9 issues where CBR fails at Top-1, but the combined TriagerX framework (CBR+IBR) correctly predicts the resolver at Top-1. We group these 43 corrected cases using the repository’s developer-assigned labels for further analysis. Table V summarizes the prevalence of each issue type. The total label count exceeds 43 as issues are multi-labeled in OpenJ9 (e.g., a *test failure* in a *VM component*).

We find that over 60% of the issues corrected by IBR

TABLE IV: Top-k accuracy of all considered baselines on different datasets compared to TriagerX CBR. P-values are calculated against the best-performing baseline (♣) where “*” indicates $p < 0.01$.

Dataset	Method	Top-1	Top-3	Top-5	Top-10	Top-20
Google Chromium	TriagerX CBR	0.345*	0.537*	0.612*	0.710*	0.803*
	DeBERTa-Large (FCN)	0.285	0.474	0.567	0.677	0.767
	RoBERTa-Large (FCN)	0.267	0.461	0.551	0.660	0.755
	BERT-Large (FCN)	0.255	0.433	0.520	0.630	0.715
	CodeBERT (FCN)	0.224	0.403	0.493	0.606	0.697
	DeBERTa-Large (CNN)	0.251	0.432	0.525	0.639	0.738
	RoBERTa-Large (CNN)	0.281	0.475	0.564	0.671	0.763
	BERT-Large (CNN)	0.271	0.455	0.549	0.655	0.743
	CodeBERT (CNN)	0.159	0.319	0.399	0.519	0.634
	LBT-P ♣	0.318	0.499	0.578	0.676	0.763
	DBRNN-A	0.183	0.318	0.385	0.482	0.581
	TF-IDF + SVM	0.204	0.310	0.376	0.454	0.529
	P-value	4.4e-18	5.7e-18	4.0e-18	5.5e-18	5.4e-18
	Mozilla Core	TriagerX CBR	0.340*	0.521*	0.598*	0.700*
DeBERTa-Large (FCN)		0.257	0.437	0.521	0.639	0.744
RoBERTa-Large (FCN)		0.276	0.458	0.540	0.650	0.749
BERT-Large (FCN)		0.215	0.378	0.461	0.571	0.681
CodeBERT (FCN)		0.206	0.371	0.455	0.570	0.678
DeBERTa-Large (CNN)		0.269	0.445	0.533	0.639	0.737
RoBERTa-Large (CNN) ♣		0.306	0.490	0.568	0.668	0.758
BERT-Large (CNN)		0.258	0.432	0.514	0.624	0.725
CodeBERT (CNN)		0.268	0.447	0.532	0.640	0.739
LBT-P		0.279	0.471	0.553	0.655	0.748
DBRNN-A		0.164	0.290	0.367	0.481	0.594
TF-IDF + SVM		0.238	0.386	0.454	0.546	0.638
P-value		1.1e-07	2.8e-06	1.9e-04	3.5e-11	7.2e-07
Mozilla Firefox		TriagerX CBR	0.272*	0.471*	0.576*	0.718*
	DeBERTa-Large (FCN)	0.221	0.402	0.488	0.646	0.801
	RoBERTa-Large (FCN)	0.218	0.400	0.505	0.642	0.781
	BERT-Large (FCN)	0.213	0.353	0.445	0.585	0.748
	CodeBERT (FCN)	0.193	0.366	0.454	0.597	0.760
	DeBERTa-Large (CNN)	0.199	0.368	0.473	0.627	0.798
	RoBERTa-Large (CNN) ♣	0.248	0.441	0.534	0.671	0.801
	BERT-Large (CNN)	0.148	0.306	0.389	0.516	0.670
	CodeBERT (CNN)	0.219	0.385	0.483	0.619	0.754
	LBT-P	0.243	0.423	0.524	0.646	0.788
	DBRNN-A	0.135	0.253	0.334	0.441	0.612
	TF-IDF + SVM	0.221	0.388	0.454	0.540	0.623
	P-value	2.3e-04	2.6e-03	4.2e-03	8.7e-03	1.0e-04

integration are test-related. These reports are typically log-heavy and template-like (e.g., stack traces and test harness output), which obscures ownership cues and causes CBR to favor developers associated with broadly similar test failures. IBR mitigates this by reordering candidates using interaction signals from similar test issues. For example, Issues #17780 and #18384 exhibit the same failure mode (*cmdLineTester_criu_nonPortableRestore*) where CBR ranked @dsouzai first, while the true resolver @JasonFengJ9 appeared lower. IBR scores enabled TriagerX to promote @JasonFengJ9 to Top-1 in both cases based on recent interactions with similar related issues.

Core Runtime issues form the second-largest group among the cases corrected by IBR. The main difficulty in these cases is that runtime components in OpenJ9 are maintained by multiple developers over time, and many runtime bug reports are textually similar to past issues that were resolved by different developers at different points in the project’s history. This makes it challenging for a content-only ranker to distinguish the most appropriate resolver based on textual similarity alone. This behavior is evident in our results—21 out of 22 runtime issues (95.5%) corrected by IBR, the true resolver already appears in CBR’s Top-5 predictions. Thus, CBR generally retrieves a relevant shortlist but often misorders candidates at rank 1 when several developers have historically resolved

TABLE V: Issue types for which IBR corrected an incorrect Top-1 prediction from CBR. Issue types are non-mutually exclusive; an issue may belong to multiple types.

Issue Type	Associated Labels	# Issues	Percentage
Test Issues	test failure, test excluded, comp:test	29	67.4%
Core Runtime	comp:vm, comp:jit, comp:jvmti, comp:ras	22	51.2%
Platform Infrastructure	comp:infra, comp:build, comp:openssl, comp:fips	12	27.9%

similar issues within the same subsystem. Analysis of OpenJ9 resolution history confirms that runtime labels reflect shared maintenance rather than exclusive ownership. For example, issues labeled `comp:jvmti` have been resolved by multiple developers, including @babsingh as well as @llxia and @thallium. When a new `jvmti` report closely resembles past reports handled by different maintainers, CBR may reasonably favor a frequent—but not *currently* most relevant resolver. For example, in issues #17871 and #18730, CBR ranks @llxia or @thallium at Top-1 while placing the true resolver @babsingh lower. IBR corrects these errors by using interaction-based evidence, using recent interactions to re-rank candidates and promote @babsingh to Top-1.

Finally, Platform Infrastructure issues include system and security-related components where issues are often resolved by the developer who is actively handling the affected infrastructure or security configuration at the time, rather than by a stable component maintainer. Consequently, textual similarity provides weak ownership signals, and CBR frequently prioritizes historically frequent contributors instead of the currently responsible developer. IBR corrects these cases by leveraging recent interaction evidence, allowing TriagerX to recover the correct resolver in 12 cases (e.g., Issues #18436, #19049).

RQ2 Summary: TriagerX CBR consistently outperforms baseline models like LBT-P and DBRNN-A, as well as larger PLMs, achieving superior Top-k accuracy across all datasets. TriagerX IBR also shows strong performance, particularly when used in conjunction with CBR within the TriagerX framework.

C. RQ3: How do multiple PLMs influence CBR accuracy?

Approach. We assess how an ensemble of multiple PLMs influences the accuracy of CBR by directly comparing it with the individual PLMs used to construct it. Concretely, we train each PLM independently using the same architecture and the same number of classifiers (i.e., 3 CNN classifiers) as in the full CBR and compare the results. We answer two sub-RQs:

RQ3.1 Overall Performance: We evaluate the performance of TriagerX CBR with individual PLMs and in ensemble to determine the impact of using multiple PLMs.

RQ3.2 Class-wise Performance: We analyze how embedding diversity from different PLMs affects class-wise Top-1 accuracy, comparing TriagerX CBR’s performance with the PLM ensemble against individual PLMs on specific classes from a benchmark dataset.

Overall Performance. The test accuracy results shown in Figure 6 indicate that both PLMs, despite being pretrained

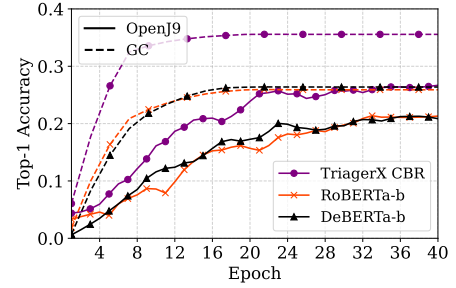


Fig. 6: Top-1 test accuracy of TriagerX CBR model compared to the RoBERTa and DeBERTa base models when trained separately with 3 classifiers on different datasets.

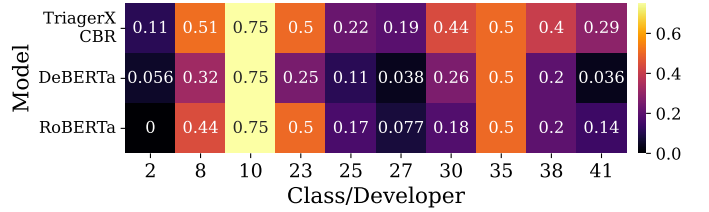


Fig. 7: Class-wise Top-1 accuracy on the OpenJ9 dataset.

on different corpora, achieve similar accuracy when fine-tuned separately on the same dataset. However, their ensemble within CBR outperforms each model, due to the complementary knowledge each PLM brings from its unique pertaining. A similar trend is observed in both the OpenJ9 and GC datasets, despite their differences in size and textual content, with accuracy improving significantly when base PLMs are ensembled. For consistency and fair evaluation, all base models were trained with identical hyperparameters, including frozen encoder layers and the number of classifiers. This confirms the effectiveness of combining multiple smaller PLMs, as the ensemble approach leverages diverse feature representations and thus enhances the robustness and generalizability of TriagerX CBR’s predictions.

Class-wise Performance. We discussed that combining multiple PLMs enhances overall performance. Here, we examine if this improvement is due to the orthogonality among models. We quantify orthogonality for PLM P_i as:

$$O(P_i) = |C_{P_i} - \left(\bigcup_{j \neq i} C_{P_j} \right)| \quad (12)$$

where C_{P_i} is the set of bug reports uniquely triaged by P_i . Higher $O(P_i)$ indicates that P_i handles unique cases well, suggesting benefits from combining PLMs. If there is a high degree of orthogonality among the base PLMs, combining them should improve class-wise Top-1 accuracy across the dataset. We compare the accuracy of TriagerX CBR using both DeBERTa and RoBERTa in ensemble versus individually on the OpenJ9 dataset. Figure 7 shows that using multiple PLMs improves Top-1 accuracy across nearly all selected classes. However, this improvement is not always linear—some classes benefit significantly (e.g., Class 8, 30, 38), while others see little to no change (e.g., Class 10, 25). This variation arises because the ensemble fuses embeddings

TABLE VI: Accuracy of different PLM combinations in CBR.

Dataset	Method	Top-1	Top-3	Top-5	Top-10	Top-20
Google Chromium	RoBERTa + DeBERTa	0.345	0.537	0.612	0.710	0.803
	RoBERTa + BERT	0.337	0.522	0.601	0.699	0.784
	DeBERTa + BERT	0.343	0.533	0.613	0.701	0.795
	CodeBERT + BERT	0.335	0.529	0.604	0.700	0.791
	CodeBERT + RoBERTa	0.342	0.529	0.611	0.703	0.791
	CodeBERT + DeBERTa	0.341	0.533	0.602	0.702	0.790
Openj9	RoBERTa + DeBERTa	0.272	0.476	0.601	0.780	0.901
	RoBERTa + BERT	0.251	0.482	0.596	0.756	0.887
	DeBERTa + BERT	0.264	0.476	0.590	0.746	0.899
	CodeBERT + BERT	0.261	0.463	0.580	0.753	0.883
	CodeBERT + RoBERTa	0.246	0.461	0.588	0.766	0.893
	CodeBERT + DeBERTa	0.259	0.481	0.588	0.748	0.891

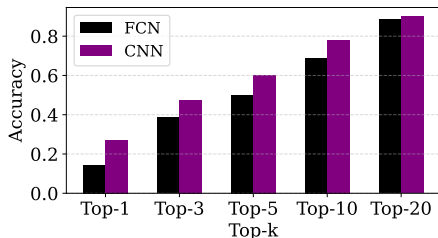


Fig. 8: Comparison of CNN vs. vanilla FCN-based classifiers.

from multiple PLMs rather than simply combining predictions. For classes where both PLMs already perform well, or where both struggle similarly (e.g., very few training data), additional embeddings may not provide further gains. Conversely, for classes where PLMs exhibit complementary strengths, the fused embeddings provide richer latent features that allow the classifier to learn more discriminative representations [41], [42]. Therefore, although some classes show little change, the consistent improvement at the aggregate level demonstrates that the PLMs capture complementary patterns that benefit the overall system.

RQ3 Summary: Combining multiple PLMs in TriagerX CBR enhances overall and class-wise accuracy. The improved performance is attributed to the orthogonality between models, resulting in higher Top-1 accuracy and better class-specific results compared to individual PLMs.

D. RQ4: Do the design decisions impact CBR performance?

Approach. We investigated three design choices affecting CBR performance: (1) Different PLM combinations, (2) replacing the CNN-based classifier with a vanilla FCN classifier, and (3) varying the number of classifiers in the TriagerX CBR model.

PLM Combinations. We evaluate multiple PLM combinations on 2 datasets. Table VI shows that all combinations yield comparable results, supporting the general robustness of our ensemble design. Among them, the RoBERTa- and DeBERTa-base pair consistently achieved slightly better performance, and thus, we adopt them as the default encoders in TriagerX. **CNN vs. Vanilla FCN.** Replacing the CNN-based classifier with a vanilla FCN in the TriagerX CBR component showed that the CNN outperforms the FCN in all Top-k metrics on the Openj9 dataset, with more than 80% improvement in Top-1 accuracy. CNN’s ability to capture contextual patterns

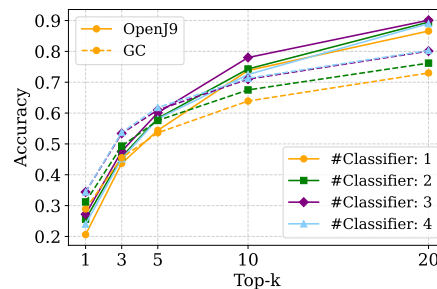


Fig. 9: Impact of the number of classifiers on Top-k accuracy across OpenJ9 and Google Chromium (GC) datasets.

from transformer hidden states contributed to its superior performance, as shown in Figure 8.

Optimal Number of Classifiers. We evaluated the effect of varying the number of classifiers in TriagerX CBR. As shown in Figure 9, performance generally improves with an increasing number of classifiers. For smaller datasets like OpenJ9, using more than 3 classifiers leads to overfitting and degraded performance. In contrast, for larger datasets like GC, increasing the number of classifiers beyond 3 yields only marginal improvement, with nearly identical Top-k accuracy curves for 3 and 4 classifiers. More classifiers also increase trainable parameters and training times. Therefore, we selected 3 classifiers for CBR across all datasets.

RQ4 Summary: The ensemble technique improves accuracy regardless of the PLM combination, CNN-based classifier consistently outperformed FCN, and using up to three classifiers provided the best balance between performance and generalizability.

E. RQ5. Do hyperparameter choices affect IBR performance?

Approach. We examine the impact of hyperparameters and configurations on IBR performance through three analyses: varying the time-decay factor λ , adjusting the similarity threshold τ , and analyzing the choice of rank aggregation (RAGg) algorithm.

Impact of Time-factored Scoring (λ). We evaluate the impact of the time-factored scoring parameter λ , with negative values prioritizing older contributions and positive values emphasizing recent ones. Figure 10 shows that negative λ values result in poorer performance, with lower Top-1, 3 accuracy scores. Conversely, positive λ values, such as $\lambda = 0.01$, significantly enhance accuracy by giving more weight to recent contributions. When $\lambda = 0$, all contributions are treated equally, leading to similar performance to CBR but less effective than weighting recent contributions more. This asserts the importance of prioritizing recent interactions for better recommendations.

Effect of Similarity Threshold (τ). We evaluate how varying the similarity threshold affects Top-1 accuracy in TriagerX. A lower threshold generally improves accuracy by including more similar issues, which increases developer interactions. However, very low thresholds increase computational demands and may introduce less relevant issues. Conversely, higher thresholds reduce the number of considered issues, potentially

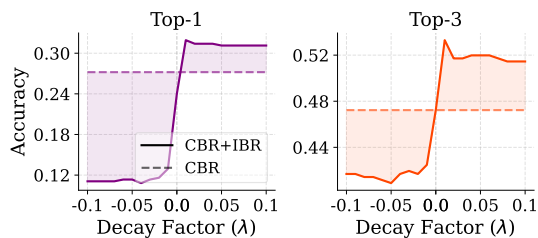


Fig. 10: Impact of time-decay factor on OpenJ9 dataset.

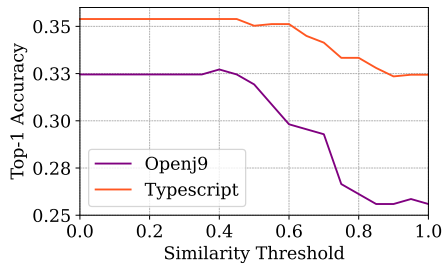


Fig. 11: Effect of similarity threshold over Top-1 accuracy.

decreasing accuracy. Figure 11 indicates that the optimal threshold in our studied datasets lies around 0.40 - 0.45.

Selection of RAgg Algorithm. We initially used the Borda count method for rank aggregation, which assigns points based on rank positions from different rankers. However, it treats all rankers equally, leading to skewed recommendation scores when developers have limited interaction with similar bugs. To address this, we employed a Weighted Ranking Aggregation (WRA) method as discussed in Section II-C. The tunable weight factor W_f in Equation 8 optimally balances IBR and CBR rankings, enhancing adaptability across datasets while keeping CBR as the primary ranker. Moreover, WRA does not affect CBR ranking when a developer lacks relevant interaction history, improving accuracy in most cases (Table VII).

Sensitivity of Fusion Weight (W_f). Figure 12 illustrates the sensitivity of Top-1 accuracy to W_f in Equation 8 across datasets with interaction history. In OpenJ9, accuracy remains stable across a broad range but peaks at $W_f = 0.65$, reflecting

TABLE VII: Comparison of Borda count ranking aggregation compared to ours.

Dataset	Aggregation Method	Top-1	Top-3	Top-5	Top-10	Top-20
Openj9	Borda Count	0.277	0.506	0.622	0.755	0.886
	WRA	0.327	0.533	0.633	0.807	0.918
TS	Borda Count	0.357	0.607	0.680	0.774	0.888
	WRA	0.353	0.615	0.711	0.830	0.930

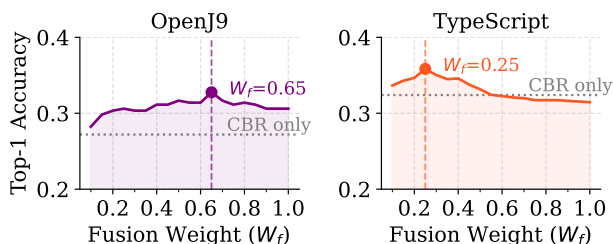


Fig. 12: Sensitivity of fusion weight W_f in different datasets.

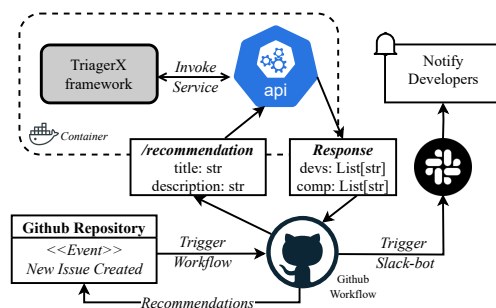


Fig. 13: Deployment pipeline of TriagerX framework.

the benefit of heavily weighting IBR signals in a repository with dense contributor interactions. In contrast, TypeScript peaks at $W_f = 0.25$ and degrades steadily as W_f increases, suggesting that over-relying on interaction signals in sparse contributor settings may negatively impact performance. In both cases, the combined framework consistently outperforms the CBR-only baseline (dotted line), demonstrating that IBR integration is broadly beneficial regardless of the exact weight chosen. For new deployments, repositories with dense interaction histories should favor higher W_f values, while sparse ones should use lower values, with final tuning via grid search (Section III-A3) on a small validation set.

RQ5 Summary: Optimizing the time-decay factor λ to favor recent contributions, choosing an optimal similarity threshold τ for balance between accuracy and computational efficiency, and employing a custom weighted rank aggregation algorithm significantly enhance the performance of TriagerX IBR.

V. INDUSTRIAL DEPLOYMENT OF TRIAGERX

We integrated TriagerX into our industrial partner’s dev environment to assess its effectiveness. At this moment, TriagerX serves triaging recommendations to the triagers for the Eclipse OpenJ9 project via GitHub Workflows. The framework runs in a Docker container for efficient deployment and maintenance, delivering recommendations via GitHub bot comments and Slack notifications.

A. TriagerX Deployment

Deployment Settings. Before deployment, we followed the same data preparation, model training, and IBR precomputation pipeline described in Sections III-A1–III-A3. The target developer set comprised the 17 most active contributors confirmed by the partner, who collectively resolved the majority of issues at the time of deployment. TriagerX is designed to support incremental retraining, allowing the model to be updated once a new developer accumulates at least 20 resolved issues, ensuring the system remains current as the contributor base evolves. More broadly, the model retraining in the industrial deployment was triggered manually on a monthly basis by the partner team, representing a low-maintenance overhead that was found to be sufficient for keeping recommendations current, given the repository’s issue resolution rate.

Technical Integration. We deployed TriagerX on a CPU-only machine—IBM Power9@2.20 GHz with 8GB memory, as this



Fig. 14: An example (#20354) of developer recommendation in the OpenJ9 Github repository.

was the hardware provided by the team. For the OpenJ9 repository, the system operates within a 4 GB memory footprint and produces component and developer recommendations in approximately 3-4 seconds, meeting the team’s real-time responsiveness needs. We also validated performance on an Intel Xeon Gold 5320@2.20 GHz CPU with the same RAM, which showed similar latency. However, when we tested under lab settings with a suitable GPU (NVIDIA V100), inference latency can be reduced to 100–200 ms. While inference is efficient on CPU, training the CBR model still requires a CUDA-enabled GPU to ensure practical training times.

The integration is orchestrated through a GitHub Actions workflow that triggers automatically whenever a new issue is opened in the repository. Upon triggering, the workflow extracts the issue title and body using the GitHub Actions context, then invokes a Node.js script that sends a POST request to the TriagerX REST API endpoint (`/recommendation`). The API processes the input and returns a ranked list of recommended developers and components, which are then posted directly as a bot comment on the issue page. Figure 13 illustrates the overall deployment pipeline, while Figure 14 shows an example where a recommended developer was subsequently assigned to the issue. The workflow was intentionally designed to be lightweight and non-intrusive; no developer onboarding or formal training sessions were required, as the bot comment format was immediately interpretable without guidance.

B. GPU Utilization of TriagerX CBR

One of the requirements from our partner was to be able to train the TriagerX models with as less memory (GPU) as possible. TriagerX CBR has fewer parameters than traditional **large** PLMs. This size reduction is achieved by combining the **base** variants of different PLM families, which, when ensembled strategically, result in a lower total parameter count compared to a typical large PLM while maintaining superior accuracy, as discussed in RQ3. For example, RoBERTa and DeBERTa base have 125M and 139M parameters, respectively. When combined in CBR, they total around 271M, which is fewer than DeBERTa-Large’s 405M parameters alone, excluding classifiers (see Appendix A). The improvement after combining the PLMs arises from the orthogonal and complementary characteristics of each model, similar to the boosting technique in AdaBoost [43], where weak learners are aggregated to strengthen predictions. Figure 15 presents a comparison of GPU memory usage, parameter count, inference

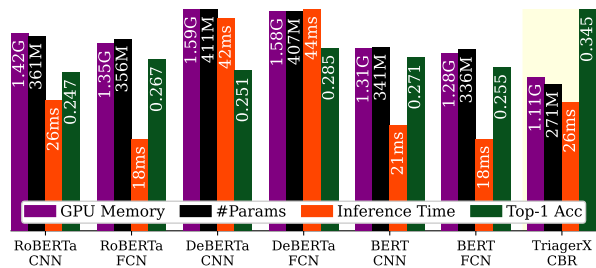


Fig. 15: Runtime GPU usage, parameters, inference time, and Top-1 accuracy of CBR vs. baselines on GC dataset.

latency, and Top-1 accuracy in the GC dataset when tested on a NVIDIA V100 GPU. For each model, we compute resource usage by averaging over 50 prediction runs, initializing a fresh runtime each time. CBR achieves higher accuracy despite using fewer parameters than larger PLM-based classifiers. Notably, the closest competitor, BERT-FCN, is approximately 15% larger in parameter size while delivering lower accuracy. In terms of latency, these models operate in the 18–44 ms range on the GPU, which we consider negligible for real-time usage. Although CodeBERT-based models offer lower memory consumption due to their smaller size, they consistently underperform in accuracy and are therefore excluded from this plot for clarity. Similarly, we omit LBT-P, a knowledge-distilled model, as TriagerX CBR does not rely on any compression techniques and thus represents a different design trade-off.

C. Component Recommendation in the Client Workflows

One initial feedback on TriagerX was that recommending component labels is also useful, especially when direct resource allocation is challenging (e.g., developer unavailability). This allows issues to be assigned to the appropriate team for resolution. Hence, we used TriagerX CBR for component classification in Openj9, focusing on all nine key components—`build`, `gc`, `infra`, `jclextensions`, `jit`, `jitserver`, `JvmTi`, `test`, `vm`. Each component belongs to a different team. We trained CBR with the same configuration as before (Section III-A2), using issue components as labels. TriagerX CBR outperforms LBT-P by 10.6% in Top-1 accuracy (Table VIII), along with large PLM-based baselines. We selected RoBERTa and DeBERTa large with CNN classifiers for comparison due to their superior performance in the developer recommendations task among other PLMs. This task benefited from a more balanced data distribution and more samples per class, improving performance compared to developer classification for all models. However, some issues, like issue #19588, have multiple component labels, impacting Top-1 accuracy, whereas Top-2 and Top-3 accuracies better represent performance in multi-label scenarios.

D. Usage Feedback from the Industrial Deployment

TriagerX is in early deployment within the OpenJ9 project, using data from the most active contributors, with feedback collection. The model was trained using historical data from

TABLE VIII: Component prediction accuracy of different models on the OpenJ9 dataset.

Model	Top-1	Top-2	Top-3
TriagerX CBR	0.782	0.928	0.971
LBT-P	0.707	0.887	0.938
RoBERTa-Large + CNN	0.734	0.901	0.961
DeBERTa-Large + CNN	0.745	0.902	0.957

TABLE IX: Top-k accuracy of TriagerX CBR and other baselines at different contribution thresholds.

Dataset	Threshold	#Dev	TriagerX CBR		LBT-P		RoBERTa-L (CNN)	
			Top-1	Top-3	Top-1	Top-3	Top-1	Top-3
Google Chromium	5	1641	0.326	0.505	0.296	0.459	0.251	0.425
	10	1320	0.329	0.513	0.309	0.484	0.262	0.438
	15	1120	0.330	0.516	0.305	0.487	0.268	0.449
	20	1032	0.345	0.537	0.318	0.499	0.281	0.475
	25	886	0.345	0.546	0.327	0.513	0.289	0.488
OpenJ9	5	103	0.232	0.423	0.181	0.304	0.131	0.264
	10	72	0.246	0.442	0.188	0.319	0.165	0.330
	15	57	0.260	0.471	0.200	0.405	0.185	0.368
	20	50	0.272	0.476	0.211	0.407	0.206	0.403
	25	44	0.281	0.486	0.230	0.412	0.209	0.411

17 active contributors. To assess user adaptation and recommendation quality, we tracked the Top-3 recommendations as per the team’s requirement for 80 new issues and measured implicit acceptance based on whether any of them participated in the issue through discussion, pull request, or commit. Using this proxy for acceptance, TriagerX achieved an 82.5% success rate. We also analyzed a subset of 51 issues that required code contributions to be resolved (e.g., commit). For 34 of these, at least one recommended developer made a code contribution, yielding a 66.7% acceptance rate.

The initial reaction from the client team on the use of TriagerX as a triaging assistant is encouraging. TriagerX provided them a valuable starting point, allowing developers to weigh the suggestions against other considerations. One developer remarked, *“This is a large repo, and I think it’s pretty good what we got—that’s a good starting point.”* The developers noted that bug triaging is complex; for example, a second developer may be assigned due to the first’s unavailability. Automated tools may overlook these contextual factors, and the recommendations could be labeled wrong; for example, a developer may be recommended based on technical similarity alone, without accounting for their current availability or workload. These contextual factors represent a direction for improvement, potentially through integrating workload or availability information into the ranking process. Another feature request on TriagerX was to explain the predictions with clear rationales. Developers expressed a desire to understand why a particular developer was recommended, such as knowing that they recently committed to several similar issues, as this would increase their confidence in accepting or overriding the suggestion, representing another direction for improvement we plan to address in future work.

VI. DISCUSSIONS

Impact of Data Imbalance. The data imbalance problem is common across the SOTA bug triaging techniques we studied. In real-world repositories, a small group of *star developers* resolve the majority of issues, causing ML/DL models to

disproportionately favor these frequent developers, reducing their effectiveness in recommending less active contributors.

Mani et al. [5] first showed that DL-based bug triaging models gain from increasing the number of training samples per developer and improvements plateau around a threshold of 20, particularly on large datasets like GC. To further examine how different approaches respond to class imbalance, we compare TriagerX CBR with two strong baselines—LBT-P and RoBERTa-Large (CNN)—under varying contribution thresholds on two datasets of very different sizes: GC and OpenJ9. As shown in Table IX, increasing the minimum contribution threshold (i.e., filtering out infrequent developers) improves accuracy across all models. However, TriagerX CBR consistently outperforms both baselines at every threshold. Finally, we use a threshold of 20 in most experiments, as it offers a good balance between predictive accuracy and developer coverage, and is consistent with prior findings.

While contribution thresholding improves predictive accuracy by reducing the impact of imbalance, it also reduces the number of developers considered, potentially excluding less experienced or newer contributors. To mitigate imbalance without aggressively narrowing coverage, we evaluated alternative techniques including oversampling, undersampling, SMOTE [44], focal loss [45], and weighted cross-entropy. Most of these approaches either degraded performance or destabilized training; weighted sampling proved the most effective and is therefore integrated into our final pipeline.

Importantly, while contribution thresholding reduces the size of the developer set, it does not substantially reduce issue coverage compared to using no threshold at all. For example, on the Google Chromium dataset, increasing the contribution threshold from 5 to 20 retains 62.8% of developers (Table IX), yet these developers still account for 91.8% of all issues relative to no thresholding. A similar trend holds for OpenJ9, where 82.0% of issues remain covered at the same threshold. This shows that a relatively small group of consistently active developers resolves most issues, and that the chosen threshold reflects a practical trade-off between coverage and reliability rather than an arbitrary exclusion.

Cold Start. Cold-start is another fundamental challenge for ML-based bug triaging approaches, as developers without prior history (e.g., new joiners) cannot be effectively modeled. In TriagerX, CBR may struggle to assign bugs to infrequent contributors when labeled training data is limited. Similarly, IBR relies on past interactions (e.g., commits) to build developer representations, which can make ranking developers with sparse/no activity more difficult.

TriagerX mitigates this challenge in two ways. First, the influence of IBR (W_f in Equation 8) can be tuned based on repository characteristics, offering flexibility for projects with sparse developer histories. Second, in collaboration with our industry partner, we extended TriagerX to also predict the component associated with an issue. Since each component is owned by a team, this recommendation functions as a team-level assignment. The corresponding team lead can then assign the issue to a suitable developer, including new joiners. This setup ensures continuity despite developer turnover and allows new developers to gradually build a contribution history. Over

time, these contributions can be incorporated into TriagerX retraining, enabling automated recommendations. While this approach does not eliminate the cold-start problem—an inherent limitation of all learning-based triaging models—it provides a practical and effective mechanism to partially mitigate it in industrial settings. For instance, consider a scenario where a new developer joins the OpenJ9 component team responsible for `comp:jit`. Without sufficient contribution history, TriagerX’s CBR and IBR components cannot directly recommend them. However, TriagerX’s component-level recommendation can correctly route the issue to the `jit` team, allowing the team lead to assign it to the new developer. Over time, as the developer resolves issues and builds a contribution history, TriagerX can incorporate their activity in subsequent retraining cycles, gradually enabling direct recommendations. Looking further ahead, the cold-start problem could be more directly addressed by leveraging pre-trained developer profiles derived from public contribution histories on platforms like GitHub, enabling the model to bootstrap representations for new developers before they accumulate sufficient local history. Similarly, transfer learning techniques could be used to adapt developer representations from repositories with rich interaction data to those with sparse histories, reducing the cold-start window for new contributors.

Threats to Validity. The **internal validity** of this study may be impacted by biases in dataset selection and preprocessing. Using different training and testing samples than LBT-P was necessary to capture our datasets’ unique characteristics but introduced discrepancies. Despite standardizing preprocessing and hyperparameters, variations in dataset handling could still lead to confounding factors. Additionally, reproducing baselines like LBT-P and MDN may result in inconsistencies, even when following the authors’ guidelines and the original paper. The **generalizability** of our findings may be constrained by the specific datasets and projects used in this study. Although we evaluated TriagerX across multiple, diverse repositories, its performance may not directly transfer to unseen projects with different developer populations, contribution patterns, or codebase characteristics. In particular, models trained on one repository are unlikely to perform well on another due to mismatches in label spaces and developer behavior. Furthermore, the validity of our industrial deployment evaluation is subject to limitations in how acceptance is measured. The acceptance rates reported in Section V-D reflect post-triage developer activity rather than direct triager acceptance, as developer engagement with an issue does not necessarily indicate that the triager was explicitly assigned based on TriagerX’s recommendation. Unobserved confounders such as developer availability, current workload, or manual override decisions may independently influence participation regardless of the recommendation. Therefore, these metrics serve as a behavioral proxy for acceptance rather than a precise measure of recommendation adoption.

VII. RELATED WORK

Broadly, automated bug triaging techniques are two types: (1) content-based and (2) developer activity-based [8], [46].

Content-based. Numerous approaches consider bug triaging as a classification problem [1], [4], [5], [7], [47] and exploit algorithms like Naïve Bayes and SVM with TF-IDF [1]–[3], [48]–[51]. Ensemble learning was also evaluated [52]. Many also approached the problem through information retrieval (IR) [53], [54]. Recent studies mostly focus on using deep learning techniques like bi-directional RNNs [5] with Word2Vec, CNNs with both context-free [4], [5] and context-sensitive [6], [7] word embedding techniques like ELMo [11] and RoBERTa, and hierarchical attention [55]. Dual-output networks are also used to predict team and developer assignments [56]. Dipongkor et al. [9] showed that different PLMs exhibit substantial orthogonality in bug triaging and suggested improvements through PLM ensembles. *Our study differs on usage of bug representations via an ensemble of PLMs that is smaller than SOTA PLMs but achieves better result than those.*

Developer Activity-based. Leveraging developer expertise scores can also improve bug triaging [57]–[59]. Methods include topic modeling as Latent Dirichlet Allocation (LDA) [60]–[64] and socio-technical models for prioritizing developers based on contributions [65]. However, these methods may struggle with ambiguous language and require extensive annotated data [66]. ML techniques using tossing graphs use developers’ tossing history [67]–[70]. IR combined with ML and DL explores developer recommendations based on code vocabulary [71], interaction metrics [21], and activity profiles [72]. Hybrid algorithms and neural networks, including CNNs [46] and ensemble approaches [73] that groups developers based on their activity, improve accuracy but these approaches also face limitations, such as reliance on specific vocabulary [71] or insufficient consideration of contextual information in bug reports [46]. *We differ in the types of interactions used and their scoring methods and by enriching the final recommendation with both content and interaction based rankings.*

VIII. CONCLUSION & FUTURE WORK

In this paper, we have presented TriagerX, a SOTA bug triaging framework that consistently outperforms recent baselines across all literature and industrial partner datasets. We do this by ensembling two novel ranking models: Content-Based Ranker (CBR) and Developer Interaction Based Ranker (IBR). Future work will focus on (1) developing a feedback-enhanced auto-adaptable bug triaging model that could learn from user feedback and improve its triaging recommendations, (2) offering a human-centric approach to use TriagerX by offering contextual explanations of recommendations using IBR’s internal signals (e.g., number of commits made by a developer on similar issues) in combination with LLM-based reasoning — such explanations could increase the confidence of human triagers and improve trust in automated decisions, and (3) extending TriagerX to related software engineering tasks such as code review and task assignment, where matching contributors to work items based on both textual content and historical activity patterns can be critical. Additionally, incorporating richer contextual signals, such as developer collaboration networks, workload history, and team structures,

TABLE X: Overview of different PLMs.

PLM	#Params	Hidden Size	Layers	Attention Heads
BERT-Base	110M	768	12	12
BERT-Large	335M	1024	24	16
RoBERTa-Base	125M	768	12	12
RoBERTa-Large	355M	1024	24	16
DeBERTa-Base	139M	768	12	12
DeBERTa-Large	405M	1024	24	16
CodeBERT	125M	768	12	12

TABLE XI: Accuracy of different graph-based approaches in our datasets.

Method	Dataset	Top-1	Top-3	Top-5	Top-10	Top-20
NCGBT	GC	0.0036	0.0098	0.0148	0.0264	0.0477
	OpenJ9	0.0353	0.1191	0.1700	0.2848	0.4437
	TS	0.0503	0.1661	0.2648	0.4255	0.7065
PCG	GC	0.0076	0.0102	0.0165	0.0293	0.0560

could further improve recommendation accuracy, particularly for repositories with complex contributor dynamics or frequent developer turnover.

APPENDIX A PLM VARIANTS

Pretrained Language Models (PLMs) are available in different sizes, offering trade-offs between performance, efficiency, and deployment requirements. Larger variants capture richer representations but require more memory and computation, making model selection dependent on task-specific needs. Table X provides an overview of the PLM variants used in this study, highlighting key architectural differences. All of the models were sourced and evaluated from the HuggingFace repository [35].

APPENDIX B ADDITIONAL BASELINES

We provide reproduction results for two recent graph-based bug triaging methods, NCGBT [39] and PCG [40], which are omitted from the main manuscript due to their inferior performance on our datasets. Both methods were reproduced using their official implementations. NCGBT performed poorly across all datasets, with performance degrading sharply as the dataset size increased. This is likely due to its reliance on CC fields, which are absent in our datasets. PCG does not depend on CC data but exhibits very low Top-k accuracy on large projects, consistent with what the original PCG paper reported for Google Chromium. Given PCG’s near-zero accuracy on the large-scale GC dataset, we did not extend its evaluation to the remaining datasets, as meaningful results were unlikely. The complete results are shown in Table XI. We provide detailed reproduction steps on these baselines in our replication package¹.

¹<https://github.com/afifaniks/triagerX>

REFERENCES

[1] G. Murphy and D. Cubranic, “Automatic bug triage using text categorization,” in *Proceedings of the sixteenth international conference on software engineering & knowledge engineering*, 2004, pp. 1–6.

[2] J. Anvik, L. Hiew, and G. C. Murphy, “Who should fix this bug?” in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE ’06. New York, NY, USA: Association for Computing Machinery, 2006, p. 361–370.

[3] J. Anvik and G. C. Murphy, “Reducing the effort of bug report triage: Recommenders for development-oriented decisions,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 20, no. 3, pp. 1–35, 2011.

[4] 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*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 926–931.

[5] S. Mani, A. Sankaran, and R. Aralikkatte, “Deeptrriage: 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*, 2019, pp. 171–179.

[6] 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.

[7] J. Lee, K. Han, and H. Yu, “A light bug triage framework for applying large pre-trained language model,” in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’22. New York, NY, USA: Association for Computing Machinery, 2023.

[8] A. Sarkar, P. C. Rigby, and B. Bartalos, “Improving bug triaging with high confidence predictions at ericsson,” in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 81–91.

[9] 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)*, 2023, pp. 1012–1023.

[10] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” in *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, 2019, pp. 4171–4186.

[11] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, “Deep contextualized word representations,” in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, M. Walker, H. Ji, and A. Stent, Eds. New Orleans, Louisiana: Association for Computational Linguistics, Jun. 2018, pp. 2227–2237.

[12] T. Mikolov, “Efficient estimation of word representations in vector space,” *arXiv preprint arXiv:1301.3781*, 2013.

[13] G. Hinton, O. Vinyals, and J. Dean, “Distilling the knowledge in a neural network,” *arXiv preprint arXiv:1503.02531*, 2015.

[14] S. D. Stanton, P. Izmailov, P. Kirichenko, A. A. Alemi, and A. G. Wilson, “Does knowledge distillation really work?” in *Advances in Neural Information Processing Systems*, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, Eds., 2021.

[15] R. T. McCoy, E. Pavlick, and T. Linzen, “Right for the wrong reasons: Diagnosing syntactic heuristics in natural language inference,” *arXiv preprint arXiv:1902.01007*, 2019.

[16] A. Elangovan, J. He, Y. Li, and K. Verspoor, “Effects of human adversarial and affable samples on bert generalization,” *arXiv preprint arXiv:2310.08008*, 2023.

[17] Y. Huang, X. Feng, B. Li, Y. Xiang, H. Wang, T. Liu, and B. Qin, “Ensemble learning for heterogeneous large language models with deep parallel collaboration,” *Advances in Neural Information Processing Systems*, vol. 37, pp. 119 838–119 860, 2024.

[18] A. Kumar Dipongkor, “An ensemble method for bug triaging using large language models,” in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, 2024, pp. 438–440.

[19] Y. Bian and H. Chen, “When does diversity help generalization in classification ensembles?” *IEEE Transactions on Cybernetics*, vol. 52, no. 9, pp. 9059–9075, 2022.

[20] L. I. Kuncheva and C. J. Whitaker, “Measures of diversity in classifier ensembles and their relationship with the ensemble accuracy,” *Machine learning*, vol. 51, no. 2, pp. 181–207, 2003.

- [21] G. Yang, T. Zhang, and B. Lee, "Utilizing a multi-developer network-based developer recommendation algorithm to fix bugs effectively," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, ser. SAC '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1134–1139.
- [22] A. Athar, S. Ali, M. M. Sheeraz, S. Bhattacharjee, and H.-C. Kim, "Sentimental analysis of movie reviews using soft voting ensemble-based machine learning," in *2021 Eighth International Conference on Social Network Analysis, Management and Security (SNAMS)*, 2021, pp. 01–05.
- [23] S. Kumari, D. Kumar, and M. Mittal, "An ensemble approach for classification and prediction of diabetes mellitus using soft voting classifier," *International Journal of Cognitive Computing in Engineering*, vol. 2, pp. 40–46, 2021.
- [24] R. M. French, "Catastrophic forgetting in connectionist networks," *Trends in cognitive sciences*, vol. 3, no. 4, pp. 128–135, 1999.
- [25] S. Ioffe and C. Szegedy, "Batch normalization: accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32nd International Conference on Machine Learning - Volume 37*, ser. ICML'15. JMLR.org, 2015, p. 448–456.
- [26] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958, 2014.
- [27] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," in *Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing (EMNLP-IJCNLP)*, 2019, pp. 3982–3992.
- [28] Y. Ding and X. Li, "Time weight collaborative filtering," in *Proceedings of the 14th ACM international conference on Information and knowledge management*, 2005, pp. 485–492.
- [29] G. Jain, T. Mahara, and S. Sharma, "Performance evaluation of time-based recommendation system in collaborative filtering technique," *Procedia Computer Science*, vol. 218, pp. 1834–1844, 2023.
- [30] J.-w. Park, M.-W. Lee, J. Kim, S.-w. Hwang, and S. Kim, "Cost-aware triage ranking algorithms for bug reporting systems," *Knowledge and Information Systems*, vol. 48, pp. 679–705, 2016.
- [31] H. Jahanshahi, M. Cevik, K. Mousavi, and A. Başar, "Adpriage: Approximate dynamic programming for bug triage," *IEEE Transactions on Software Engineering*, vol. 49, no. 10, pp. 4594–4609, 2023.
- [32] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," *arXiv preprint arXiv:1711.05101*, 2017.
- [33] K. Song, X. Tan, T. Qin, J. Lu, and T.-Y. Liu, "Mpnnet: Masked and permuted pre-training for language understanding," *Advances in neural information processing systems*, vol. 33, pp. 16857–16867, 2020.
- [34] Y. Kaya, S. Hong, and T. Dumitras, "Shallow-deep networks: Understanding and mitigating network overthinking," in *International conference on machine learning*. PMLR, 2019, pp. 3301–3310.
- [35] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz *et al.*, "Huggingface's transformers: State-of-the-art natural language processing," *arXiv preprint arXiv:1910.03771*, 2019.
- [36] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.
- [37] P. He, X. Liu, J. Gao, and W. Chen, "Deberta: Decoding-enhanced bert with disentangled attention," *arXiv preprint arXiv:2006.03654*, 2020.
- [38] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, T. Cohn, Y. He, and Y. Liu, Eds. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547.
- [39] 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*, vol. 235, p. 103093, 2024.
- [40] J. Dai, Q. Li, S. Xie, D. Li, and H. Chu, "Pcg: A joint framework of graph collaborative filtering for bug triaging," *Journal of Software: Evolution and Process*, vol. 36, no. 9, p. e2673, 2024.
- [41] B. Lester, D. Pressel, A. Hemmeter, S. R. Choudhury, and S. Bangalore, "Multiple word embeddings for increased diversity of representation," *arXiv preprint arXiv:2009.14394*, 2020.
- [42] H. Tsukagoshi, R. Sasano, and K. Takeda, "Comparison and combination of sentence embeddings derived from different supervision signals," in *Proceedings of the 11th Joint Conference on Lexical and Computational Semantics*, V. Nastase, E. Pavlick, M. T. Pilehvar, J. Camacho-Collados, and A. Raganato, Eds. Seattle, Washington: Association for Computational Linguistics, Jul. 2022, pp. 139–150.
- [43] R. E. Schapire, "Explaining adaboost," in *Empirical inference: festschrift in honor of vladimir N. Vapnik*. Springer, 2013, pp. 37–52.
- [44] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.
- [45] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2980–2988.
- [46] 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 Process. Lett.*, vol. 51, no. 3, p. 2589–2606, jun 2020.
- [47] W. Zhang, "Efficient bug triage for industrial environments," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 727–735.
- [48] S. N. Ahsan, J. Ferzund, and F. Wotawa, "Automatic software bug triage system (bts) based on latent semantic indexing and support vector machine," in *2009 Fourth International Conference on Software Engineering Advances*, 2009, pp. 216–221.
- [49] J. Kanwal and O. Maqbool, "Bug prioritization to facilitate bug report triage," *Journal of Computer Science and Technology*, vol. 27, pp. 397–412, 2012.
- [50] V. Dedík and B. Rossi, "Automated bug triaging in an industrial context," in *2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEEA)*, 2016, pp. 363–367.
- [51] W. Fu and T. Menzies, "Easy over hard: a case study on deep learning," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 49–60.
- [52] L. Jonsson, M. Borg, D. Broman, K. Sandahl, S. Eldh, and P. Runeson, "Automated bug assignment: Ensemble-based machine learning in large scale industrial contexts," *Empirical Software Engineering*, vol. 21, no. 4, pp. 1533–1578, 2016.
- [53] M. Linares-Vásquez, K. Hossen, H. Dang, H. Kagdi, M. Gethers, and D. Poshyvanik, "Triageing incoming change requests: Bug or commit history, or code authorship?" in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, 2012, pp. 451–460.
- [54] R. Shokripour, J. Anvik, Z. M. Kasirun, and S. Zamani, "Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation," in *2013 10th Working Conference on Mining Software Repositories (MSR)*, 2013, pp. 2–11.
- [55] H. He and S. Yang, "Automatic bug triage using hierarchical attention networks," in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 2021, pp. 1043–1049.
- [56] C. A. Choquette-Choo, D. Sheldon, J. Proppe, J. Alphonso-Gibbs, and H. Gupta, "A multi-label, dual-output deep neural network for automated bug triaging," in *2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)*, 2019, pp. 937–944.
- [57] A. Tamrawi, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen, "Fuzzy set and cache-based approach for bug triaging," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 365–375.
- [58] H. Hu, H. Zhang, J. Xuan, and W. Sun, "Effective bug triage based on historical bug-fix information," in *2014 IEEE 25th International Symposium on Software Reliability Engineering*, 2014, pp. 122–132.
- [59] Y. Tian, D. Wijedasa, D. Lo, and C. Le Goues, "Learning to rank for bug report assignee recommendation," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, 2016, pp. 1–10.
- [60] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of machine Learning research*, vol. 3, no. Jan, pp. 993–1022, 2003.
- [61] X. Xie, W. Zhang, Y. Yang, and Q. Wang, "Dretom: developer recommendation based on topic models for bug resolution," in *Proceedings of the 8th International Conference on Predictive Models in Software Engineering*, ser. PROMISE '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 19–28.
- [62] T. Zhang, G. Yang, B. Lee, and E. K. Lua, "A novel developer ranking algorithm for automatic bug triage using topic model and developer relations," in *2014 21st Asia-Pacific Software Engineering Conference*, vol. 1, 2014, pp. 223–230.
- [63] G. Yang, T. Zhang, and B. Lee, "Towards semi-automatic bug triage and severity prediction based on topic model and multi-feature of bug reports," in *2014 IEEE 38th Annual Computer Software and Applications Conference*, 2014, pp. 97–106.

- [64] X. Xia, D. Lo, Y. Ding, J. M. Al-Kofahi, T. N. Nguyen, and X. Wang, "Improving automated bug triaging with specialized topic model," *IEEE Transactions on Software Engineering*, vol. 43, no. 3, pp. 272–297, 2016.
- [65] J. Xuan, H. Jiang, Z. Ren, and W. Zou, "Developer prioritization in bug repositories," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 25–35.
- [66] W. Zhang, S. Wang, Y. Yang, and Q. Wang, "Heterogeneous network analysis of developer contribution in bug repositories," in *2013 International Conference on Cloud and Service Computing*, 2013, pp. 98–105.
- [67] 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.
- [68] P. Bhattacharya and I. Neamtiu, "Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging," in *2010 IEEE International Conference on Software Maintenance*, 2010, pp. 1–10.
- [69] S. Xi, Y. Yao, X. Xiao, F. Xu, and J. Lu, "Bug triaging based on tossing sequence modeling," *Journal of Computer Science and Technology*, vol. 34, pp. 942 – 956, 2019.
- [70] Y. Su, Z. Xing, X. Peng, X. Xia, C. Wang, X. Xu, and L. Zhu, "Reducing bug triaging confusion by learning from mistakes with a bug tossing knowledge graph," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 191–202.
- [71] D. Matter, A. Kuhn, and O. Nierstrasz, "Assigning bug reports using a vocabulary-based expertise model of developers," in *2009 6th IEEE International Working Conference on Mining Software Repositories*, 2009, pp. 131–140.
- [72] H. Naguib, N. Narayan, B. Brügge, and D. Helal, "Bug report assignee recommendation using activity profiles," in *2013 10th Working Conference on Mining Software Repositories (MSR)*, 2013, pp. 22–30.
- [73] W. Zhang, J. Zhao, and S. Wang, "Sustrriage: sustainable bug triage with multi-modal ensemble learning," in *IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, 2021, pp. 441–448.