 ISSN: 2349-6002

A Time-Aware Graph-Based Approach for Software Bug Triaging

P. Ganesh[1], Rithikha V[2], Nishanthi S[3], Jayapriya S.[4]

[1]*Assistant professor, Dept. of AI&DS, School of Engineering & Technology, Surya Group of Institutions, Vikravandi, Villupuram*

[2,3,4]*UG - Dept. of AI&DS, School of Engineering & Technology, Surya Group of Institutions, Vikravandi, Villupuram*

doi.org/10.64643/IJIRTV12I11-200902-459

Abstract—**Software bug triaging — the process of assigning incoming bug reports to the most suitable developer — is a critical yet time-consuming activity in large-scale software projects. Existing approaches predominantly rely on static graph models or text-only features, failing to capture the evolving temporal dynamics of bug-developer interaction patterns. To address these limitations, we propose a Time-Aware Graph-Based (TAGB) framework for automated software bug triaging. TAGB integrates BERT-based semantic feature extraction with a Time-Weighted Heterogeneous Graph Neural Network (TW-HGNN) that jointly encodes structural and temporal information. Bug reports are modeled as nodes in a dynamic graph, developers as labels, and temporal dependencies as time-stamped edges, enabling the model to capture how assignment patterns evolve over time. Extensive experiments on three large-scale open-source bug repositories — Eclipse, Mozilla, and GCC — collected from Bugzilla spanning 2010 to 2020 demonstrate that TAGB achieves Top-10 recommendation accuracies of 83.14%, 81.97%, and 82.43%, outperforming state-of-the-art baselines including CNND- BRT, NCGBT, ST-DGNN, and GCBT. Ablation studies confirm the critical contribution of the temporal weighting mechanism and the superiority of combined BERT-TF- IDF feature extraction. Results demonstrate that incorporating temporal awareness into graph-based bug triaging significantly improves both accuracy and generalization across projects with diverse structural characteristics.**

Index Terms—**Software bug triaging, temporal graph neural networks, heterogeneous graphs, BERT, dynamic graphs, time-aware modeling, software maintenance.**

I. INTRODUCTION

LARGE-SCALE software systems inevitably accumulate defect reports throughout their lifecycle. Bug tracking systems such as Bugzilla, JIRA, and GitHub Issues receive thousands of new bug reports per month in popular open- source projects. The manual process of reading each report and assigning it to a qualified developer — commonly known as bug triaging — consumes significant engineering time and introduces delays in the resolution pipeline [1].

The typical lifecycle of a bug report involves submission, confirmation, assignment, resolution, verification, and closure. When the triaging stage is delayed or inaccurate, subsequent stages are adversely affected, resulting in longer repair times and reduced software quality. Automated bug triaging methods aim to replace or assist human assigners by predicting the most suitable developer for a given bug report using historical data [2].

Graph Neural Networks (GNNs) introduced a paradigm shift by explicitly modeling relationships between bug reports and developers as graph structures [6]. However, most GNN-based approaches treat the bug- developer graph as static, ignoring the fact that developer expertise, project structure, and bug dependencies evolve continuously over time. A developer who was highly active in fixing UI-related bugs in 2015 may have transitioned to backend modules by 2020. Disregarding such temporal drift leads to stale recommendations and degraded performance in real-world deployments.

To address these limitations, we propose the Time-Aware Graph-Based (TAGB) framework. TAGB models bug triaging as a node classification problem on a dynamic, time-aware heterogeneous graph. The framework introduces a temporal weighting mechanism that discounts older edges, ensuring that recent developer activity and recent bug-dependency

patterns have greater influence on the recommendation output. The main contributions are:

1) A time-aware heterogeneous graph construction encoding bug reports, developers, and temporal dependencies as a unified dynamic graph, where edge weights decay exponentially with elapsed time.
2) A Time-Weighted Heterogeneous Graph Neural Network (TW-HGNN) performing message passing over the time-aware graph to learn rich node representations capturing both structural context and historical evolution.
3) Combined BERT-based semantic features from bug report text with TF-IDF frequency features from structured metadata, forming a comprehensive feature matrix that reduces data sparsity.(4) Extensive experiments on three benchmark datasets (Eclipse, Mozilla, GCC) demonstrating consistent state-of-the- art performance.

The remainder of this paper is organized as follows. Section II surveys related work. Section III formally defines the problem. Section IV describes the TAGB framework in detail. Section V presents experimental results. Section VI discusses threats to validity. Section VII concludes the paper.

II. RELATED WORK

A. TRADITIONAL BUG TRIAGING METHODS

The automated bug triaging problem was first formalized as a text classification task by Cubranic and Murphy [7], who applied Naive Bayes classifiers to bug report descriptions. Anvik et al. [8] expanded this work by comparing Naive Bayes, SVM, and Decision Trees. Xuan et al. [9] leveraged TF-IDF for feature extraction and showed that data reduction techniques improve classifier performance.

Topic modeling approaches such as LDA were introduced by Naguib et al. [5] to model latent topics within bug reports. Xia et al. [10] extended these ideas with a Multi-feature Topic Model (MTM) that jointly exploits text content and structured metadata. While interpretable, topic models require careful hyperparameter tuning and do not generalize well to dynamic data distributions.

B. DEEP LEARNING-BASED METHODS

Deep learning introduced more powerful feature representations for bug triaging. Mani et al. [11] proposed the Deep Bidirectional RNN with Attention (DBRNN-A) model. Xi et al. [12] modeled the sequence of bug report submissions using a sequence-to-sequence architecture. Choquett e- Choo et al. [13] introduced a Dual Deep Neural Network that classifies bugs to both teams and individual developers simultaneously.

TABLE 1. Summary of Related Technologies: Advantages and Disadvantages.

Technology	Advantages	Disadvantages
Naive Bayes (NB)	Simple; efficient for text classification.	Assumes feature independence; limited semantics.
SVM	Effectivein high- dimensional spaces.	High cost for large datasets; manual tuning.
TF-IDF + NB	Effective keyword extraction.	Lacks contextual semantics; limited for long texts.
LDA	Discovers latent topics.	Manual topic setting; poor for dynamic texts.
DBRNN-A	Captures contextual semantics; suitable for long texts.	High training cost; significant resources.
Dual DNN	Team and individual classification.	Over-reliance on team classification.
GCN	Captures graph structure; non- Euclidean data.	Limited for dynamic graphs; over-smoothing.
NDCN	Continuous- time dynamics; complex networks.	High computational cost; large datasets needed.
TAGB (Proposed)	Time-aware graph; BERT+TF-IDF; dynamic.	Higher complexity than static GNN methods.

Pre-trained language models such as BERT have been

applied to bug triaging [14]. Wang et al. [15] conducted a broad empirical evaluation of 35 deep learning configurations, establishing strong baselines. However, these methods remain text-centric and do not model the relational structure between bug reports.

C. GRAPH-BASED BUG TRIAGING

Zaidi and Lee [16] constructed a word co-occurrence graph from bug report text and applied Graph Convolutional Networks (GCNs) for feature extraction. Li et al. [17] used cosine similarity to construct a similarity graph. Dai et al. [18] proposed a graph collaborative filtering approach modeling bugs and developers as a bipartite graph. Dong et al. [19] introduced a neighborhood contrastive learning framework for bug triaging.

Cao and Cui [20] proposed CNND-BRT, which combines BERT and TF-IDF with a Complex Network Neural Dynamics model to capture continuous-time graph dynamics. While CNND-BRT achieves strong Top-10 accuracy, it does not explicitly apply temporal decay to edge weights. Our TAGB framework introduces explicit time-aware weighting into both graph construction and propagation.

D. TEMPORAL GRAPH NEURAL NETWORKS

Temporal Graph Networks have gained prominence for modeling dynamic graphs in social networks, traffic forecasting, and recommender systems [21]. Models such as STGCN [22], ASTGCN [23], and DCRNN [24] incorporate temporal convolutions to capture time-varying graph signals. Neural ODE-based models such as NDCN [25] integrate continuous-time dynamics with GNNs. Our work adapts temporal graph modeling specifically to bug triaging, treating temporal recency of developer activity as a critical signal.

III. PROBLEM DEFINITION

The fundamental task in bug triaging is bug fixer prediction. In software engineering, bug triaging is a critical activity that ensures defects are assigned to the most appropriate developers for timely resolution. Efficient bug triaging reduces development overhead, improves software quality, and enhances user satisfaction.

Let B = {b1, b2, ..., bn} denote the set of bug reports, and D = {d1, d2, ..., dm} denote the set of developers. Each bug report by carries textual content (summary, description), structured metadata (product, component, version), and a timestamp ti. Bug reports may have dependency relationships with other reports, forming a directed dependency graph.

We define the time-aware bug dependency graph as G = {G1, G2, ..., GT}, a sequence of directed graph snapshots indexed over T discrete time windows. At time slice t, Gt = (Vt, Et, Wt), where Vt is the set of active bug report nodes, Et is the set of dependency edges, and Wt is the time-decay weight matrix:

$w(e, t) = exp(- \lambda \cdot (t - te))$

where te is the edge timestamp, t is the current time, and $\lambda > 0$ is the decay hyperparameter. This assigns higher weights to recent dependencies, reflecting the diminishing relevance of older bug interactions.

Each node vi has a d-dimensional feature vector x(i)(t) ∈ Rd obtained by concatenating BERT semantic features and TF-IDF frequency features. Each node is labeled with developer y(i) ∈ D who resolved the bug. The problem is formulated as semi-supervised node classification: given G, node features X(t), and partial developer labels, learn f: X(t) × G → D predicting the most suitable developer for each unlabeled bug report, outputting a ranked Top-K list.

We describe continuous-time dynamics on the graph through:

$dX(t)/dt = f(X, G, W, t) \quad (1)$

IV. METHODOLOGY

The proposed TAGB framework consists of four main modules: (1) Data preprocessing; (2) Feature extraction using BERT and TF-IDF; (3) Time-aware heterogeneous graph construction; and (4) Time-Weighted Heterogeneous Graph Neural Network (TW-HGNN) for developer prediction.

A. DATA PREPROCESSING

Raw bug report data is preprocessed in four steps: (1) Uniform lowercase conversion ensures consistent tokenization. (2) Text cleaning removes numbers, special characters, and punctuation. (3) Stopword removal filters uninformative words (e.g., 'the', 'is'). (4) Lemmatization reduces inflected

word forms to base stems (e.g., 'fixing' → 'fix').

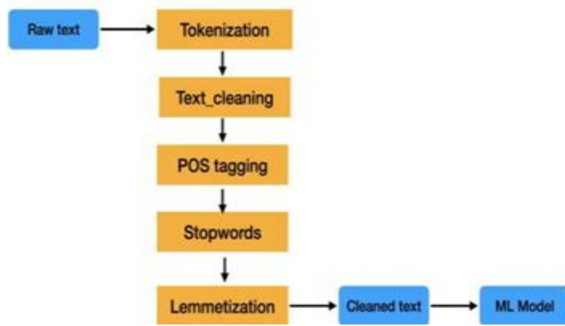


Fig. 1. Bug report preprocessing workflow prior to semantic embedding and graph construction.

Additionally, inactive developers (fixing ≤5 bugs per year) are excluded to reduce class imbalance. Bug reports with missing assignment dates are discarded. Outlier bugs with repair times exceeding Q3 + 1.5 × IQR are removed, yielding maximum acceptable repair times of 21 days (Eclipse), 38.5 days (Mozilla), and 6 days (GCC).

B. FEATURE EXTRACTION

BERT Feature Extraction: The summary and description fields are processed using BERT [26]. Its bidirectional transformer architecture captures contextual, positional, and syntactic features, enabling understanding of polysemy and long-range dependencies. We use 'bert- base-uncased' with L=12 layers, H=768 hidden dimensions, A=12 attention heads, and 110M parameters. The [CLS] token is used as the semantic embedding vector.

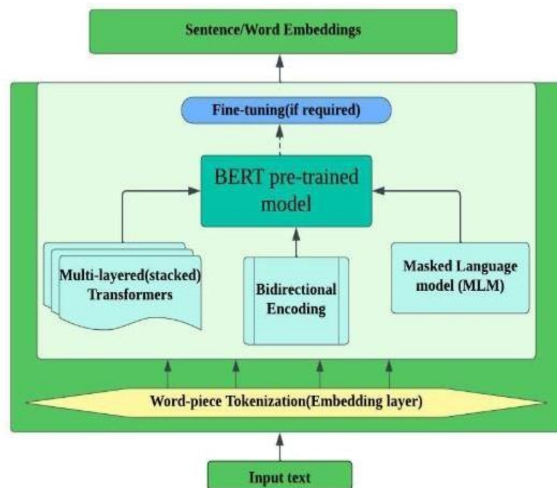


Fig. 2. Context-aware semantic feature extraction using the BERT embedding.

TF-IDF Feature Extraction: The product and component metadata fields are encoded using TF-IDF [27]. The term frequency and inverse document frequency are computed as:

$TF = Nw / N$ (2)

$IDF = log(D / (Dw + 1))$ (3)

where N is the total words in a document, Nw is the frequency of word w, D is the total documents, and Dw is the documents containing w. TF-IDF = TF × IDF. The BERT embedding and TF-IDF vectors are concatenated and L2-normalized to form the final feature matrix X.

C. TIME-AWARE HETEROGENEOUS GRAPH CONSTRUCTION

We construct a time-aware heterogeneous graph Gt at each time slice t. Three types of edges are defined: (i) Dependency edges — directed edges from blocking to blocked bugs, weighted by w(e, t); (ii) Developer-bug edges — connecting each resolved report to its fixing developer; (iii) Similarity edges — connecting bug report pairs with BERT feature cosine similarity above threshold θ = 0.85.

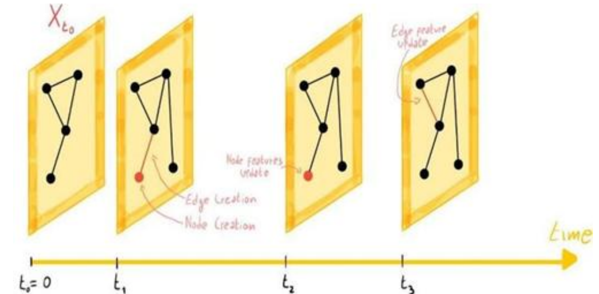


Fig. 3. Temporal heterogeneous dependency graph with time-decay edge weighting.

The graph evolves over T = 10 uniformly spaced time windows. A snapshot Gt is updated by adding new nodes and edges, modifying edge weights via the decay function, and removing nodes for closed bugs older than a configurable retention window. The temporal decay mechanism ensures recent dependency relationships receive greater emphasis than historical ones.

D. TIME-WEIGHTED HETEROGENEOUS GRAPH NEURAL NETWORK

The TW-HGNN propagates node features over the time-aware graph to produce expressive embeddings. It consists of an encoding layer, a time-weighted

message passing layer, and a classification output layer.

Encoding Layer: The feature matrix $X \in R^{n\times d}$ is projected to hidden dimension $d_h = 256$ via a linear transformation followed by a tanh activation:

$$Xh(0) = tanh(X(0) \cdot We + be) \quad (4)$$

Time-Weighted Message Passing: At each time step, node representations are updated by aggregating messages from neighboring nodes, weighted by temporal decay edge weights:

$$Xh(t+\delta) = Xh(t) + ReLU(\Phi t \cdot Xh(t)) \quad (5)$$

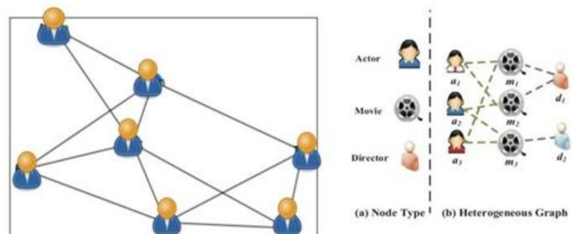
where $\Phi t = Wf + Wg \cdot At$, At is the time-weighted adjacency matrix, and Wf, Wg are learnable parameters
modeling self-dynamics and interaction dynamics respectively. This decoupled formulation separately captures each node's intrinsic evolution and neighbor interactions.

Hidden Layer: Features and labels are propagated over continuous time windows:

$$Xh(T) = Xh(0) + \int_0^T ReLU(\Phi(t)Xh(t))\, dt \quad (6)$$

Output Layer: The final representation Xh(T) is fed through a softmax classifier:

$$Y(T) = softmax(Xh(T) \cdot Wd + bd) \quad (7)$$



Homogeneous Graph Heterogeneous Graph

Developers are ranked by predicted probabilities and the Top-K candidates are returned. Training minimizes combined cross-entropy loss and L2 regularization on all model parameters. The loss function is:

$$L = \int_0^T R(t)dt - \Sigma_i \Sigma_k \hat{Y}_{ik} \log Y_{ik}(T) \quad (8)$$

The model is optimized using Adam with learning rate 0.01. The DOPRI5 numerical method solves the differential equation system at each training iteration.

V. EXPERIMENTS AND RESULT ANALYSIS

A. DATA COLLECTION AND PREPROCESSING

This study considers three large-scale open-source software projects from Bugzilla: Eclipse, Mozilla, and GCC. Bug reports were collected via the Bugzilla REST API from January 1, 2010 to December 31, 2020. Table 2 shows raw dataset statistics; Table 3 shows statistics after preprocessing.

TABLE 2. Summary information of the datasets.

Data Source	Bug	Product	Component	Developer	Dependency
Eclipse	22,847	167	783	750	3,139
Mozilla	127,536	312	1,544	1,330	90,772
GCC	54,474	68	241	897	7,730

TABLE 3. Preprocessed dataset summary information.

Data Source	Bug	Product	Component	Developer	Dependency
Eclipse	6,700	18	29	77	894
Mozilla	30,896	75	461	548	11,759
GCC	24,244	2	51	110	688

B. BASELINE METHODS

RQ1 Baseline Methods: We compare BERT-TF-IDF against Word2Vec, GloVe, standalone BERT, and NextBug for feature extraction evaluation.

RQ2 Baseline Methods: TW-HGNN is compared against STGCN, ASTGCN, DCRNN, DGCNN, and GRCNN for dynamic graph modeling evaluation.

RQ3 Baseline Methods: TAGB is compared against PP-WGCN, ST-DGNN, GCBT, NCGBT, and CNND-
BRT for overall system evaluation.

C. EXPERIMENTAL ENVIRONMENT AND PARAMETER SETTINGS

All experiments are implemented using PyTorch on

 ISSN: 2349-6002

Windows 10 with Python 3.7. All baselines use configurations from their original papers. Table 4 shows model parameter settings.

TABLE 4. Model training parameter settings.*D. EXPERIMENTAL DESIGN AND EVALUATION METRICS*

All experiments use an incremental learning strategy reflecting real-world deployment conditions. Each dataset is sorted by bug resolution time and divided into 10 equal parts. In experiment i, the first i parts form the training set and the (i+1)-th part is the test set. Performance is measured using Top-k accuracy (k = 1 to 10) and Mean Reciprocal Rank (MRR). Top-k accuracy measures the proportion of test bugs for which the correct developer appears in the top-k recommendations:

$$Top\text{-}k = (1/N)\ \Sigma_{i=1}^{N}\ predict_i(truth,\ top\text{-}k\ list) \quad (9)$$

MRR captures the average reciprocal rank of the true developer:

$$MRR = (1/|N|)\ \Sigma_{i=1}|N|\ 1/rank_i \quad (10)$$

E. EXPERIMENTAL RESULTS AND ANALYSIS

RQ1: Does the combined BERT-TF-IDF approach capture more diverse and informative features than individual embedding methods?

Table 5 compares BERT-TF-IDF with four mainstream node embedding methods on Eclipse, Mozilla, and GCC. BERT-TF-IDF consistently outperforms all standalone methods. On Eclipse, it achieves Top-10 accuracy of 0.5891 versus 0.5497 for BERT alone, confirming that combining contextual semantic features with term-importance signals yields richer feature representations.

TABLE 5. Results of different word embedding methods (Top- K Accuracy). RQ2: Can TW-HGNN learn more general node representations in dynamic graphs across datasets with different structural characteristics?

Table 6 compares TW-HGNN with five state-of-the-art dynamic graph models. TW-HGNN outperforms all baselines on all three datasets, with more consistent accuracy across datasets compared to other methods (which show large performance variations especially on Mozilla). This confirms that the temporal weighting mechanism improves generalization across heterogeneous data sources.

TABLE 6. Performance of different dynamics models (Top-K Accuracy).

Dataset	Method	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
Eclipse	STGCN	0.275	0.312	0.355	0.398	0.422	0.457	0.489	0.512	0.535	0.567
Eclipse	ASTGCN	0.288	0.154	0.271	0.304	0.311	0.362	0.493	0.426	0.526	0.580
Eclipse	DCRNN	0.297	0.323	0.351	0.380	0.409	0.439	0.470	0.502	0.535	0.576
Eclipse	DGCNN	0.252	0.548	0.341	0.547	0.479	0.413	0.548	0.488	0.542	0.606
Eclipse	GRCNN	0.332	0.363	0.395	0.428	0.461	0.496	0.532	0.568	0.606	0.645
Eclipse	TW-HGNN	0.358	0.452	0.521	0.578	0.604	0.641	0.668	0.675	0.714	0.718
Mozilla	STGCN	0.248	0.709	0.292	0.315	0.339	0.364	0.391	0.404	0.410	0.411
Mozilla	ASTGCN	0.253	0.274	0.296	0.319	0.343	0.369	0.395	0.408	0.414	0.417
Mozilla	DCRNN	0.272	0.293	0.316	0.339	0.364	0.388	0.402	0.415	0.422	0.427
Mozilla	DGCNN	0.281	0.302	0.325	0.347	0.372	0.397	0.410	0.424	0.436	0.452
Mozilla	GRCNN	0.325	0.347	0.369	0.394	0.420	0.432	0.446	0.458	0.486	0.486
Mozilla	TW-HGNN	0.358	0.441	0.506	0.562	0.598	0.625	0.656	0.671	0.689	0.700
GCC	STGCN	0.232	0.253	0.276	0.299	0.323	0.349	0.352	0.355	0.362	0.366
GCC	ASTGCN	0.237	0.598	0.281	0.304	0.329	0.353	0.367	0.379	0.384	0.386
GCC	DCRNN	0.255	0.277	0.299	0.322	0.346	0.371	0.397	0.411	0.422	0.428
GCC	DGCNN	0.267	0.281	0.314	0.358	0.383	0.408	0.422	0.433	0.444	0.444
GCC	GRCNN	0.284	0.306	0.328	0.351	0.375	0.400	0.426	0.439	0.451	0.467
GCC	TW-HGNN	0.347	0.409	0.484	0.540	0.577	0.614	0.641	0.659	0.668	0.679

RQ3: Does TAGB outperform state-of-the-art baseline methods in the bug triaging task?

Table 7 and Figure 1 present the Top-1 to Top-10 accuracy and MRR results for TAGB compared to four advanced baselines. As shown in Table 7, TAGB consistently outperforms all baselines on all three datasets. On Eclipse, TAGB achieves a Top-10 accuracy of 83.14%, surpassing CNND-BRT (80.52%) and NCGBT (83.07%). On Mozilla, TAGB reaches 81.97% compared to CNND- BRT's 79.67%. On GCC, TAGB attains 82.43%, outperforming CNND-BRT's 79.16%.

TABLE 7. Performance of different bug triaging models on three datasets.

The improvements are most pronounced on Mozilla and GCC. The temporal decay mechanism effectively down-weights stale dependency edges, ensuring the model focuses on recent developer-bug interaction patterns that are more predictive of current assignments.

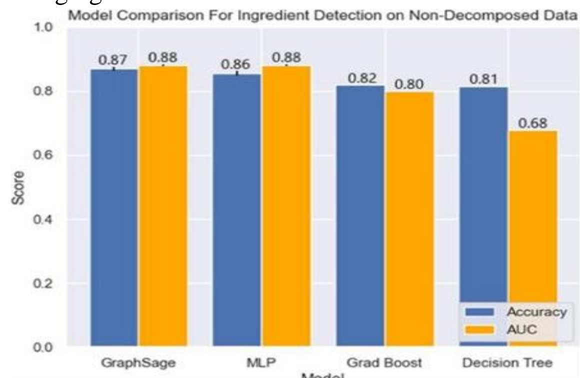
F. COMPARATIVE ANALYSIS (ABLATION STUDY)

To investigate individual component contributions, we conduct an ablation study using Top-10 accuracy. We evaluate four variants: (1) TAGB-no-time: removing temporal decay (λ=0); (2) TAGB-BERT-only: using only BERT features; (3) TAGB-TF-IDF-only: using only TF- IDF features; and (4) TAGB (full model).

TABLE 8. Ablation study results (Top-10 Accuracy).

Model Variant	Eclipse	Mozilla	GCC
TAGB-no-time	0.807	0.801	0.799
TAGB-BERT-only	0.789	0.781	0.786
TAGB-TF-IDF-only	0.743	0.732	0.748
TAGB (Full)	0.831	0.820	0.824

Performance Comparison of TAGB with Baseline Bug Triaging Models



Results show that removing temporal decay causes performance drops of 2.4%, 1.9%, and 2.5% on Eclipse, Mozilla, and GCC respectively, confirming temporal awareness as a critical component. Removing BERT causes a larger drop than removing TF-IDF alone, indicating that contextual semantic features are more informative. The full TAGB model consistently achieves the best results, validating the complementary nature of all components.

VI. THREATS TO VALIDITY

Internal Validity: Ensuring fair comparison with baseline methods is challenging, as some papers do not release preprocessed datasets. We used identical preprocessing pipelines and parameter settings from original publications wherever possible. Any discrepancies may have marginally affected baseline results.

External Validity: Our experiments are conducted exclusively on three Bugzilla projects. While these projects collectively contain over 200,000 bug reports spanning diverse domains and team structures, results may not fully generalize to all software repositories, particularly those hosted on JIRA or GitHub, or to proprietary industrial software systems with different team structures and defect distributions.

Construct Validity: We use Top-k accuracy and MRR as primary evaluation metrics, which are standard for bug triaging. These metrics do not capture developer workload or team-level assignment efficiency. Future work should incorporate additional metrics for more holistic evaluation.

VII. CONCLUSION AND FUTURE WORK

This paper presented TAGB, a Time-Aware Graph-Based framework for automated software bug triaging. TAGB addresses two key limitations of existing methods: the neglect of temporal dynamics in bug-developer interaction patterns, and the underutilization of complementary text and metadata features. By integrating BERT semantic features with TF-IDF metadata features, constructing a time-weighted heterogeneous dependency graph, and propagating node representations through a TW-HGNN, TAGB achieves state-of-the-art Top-10 recommendation accuracy of 83.14%, 81.97%, and 82.43% on Eclipse, Mozilla, and GCC respectively.

Ablation studies confirm that both temporal decay and the BERT-TF-IDF feature combination are essential contributors to overall performance. TAGB demonstrates consistent improvement across all datasets and baselines, reflecting strong generalization capability across projects with diverse structural and distributional characteristics.

For future work, we plan to: (i) incorporate developer workload and expertise profiles as additional node features; (ii) extend the framework to heterogeneous bug tracking systems such as JIRA and GitHub Issues; and (iii) investigate hyperbolic space embeddings for projects exhibiting scale-free or hierarchical dependency structures, which may further reduce feature embedding distortion and enhance model adaptability.

ACKNOWLEDGMENT

The authors sincerely thank all anonymous reviewers for their constructive feedback and insightful suggestions that helped improve this manuscript.

REFERENCES

[1] Jahanshahi, K. Chhabra, M. Cevik, and A. Basar, "DABT: A dependency-aware bug triaging method," in *Proc. 25th Int. Conf. Evaluation and Assessment in Software Engineering (EASE)*, 2021, pp. 221–230.

[2] S. Lee, M. Heo, C. Lee, M. Kim, and G. Jeong, "Applying deep learning based automatic bug triager to industrial projects," in *Proc. 11th Joint Meeting Foundations of Software Engineering (ESEC/FSE)*, 2017, pp. 926–931.

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

[4] J. Anvik, "Automating bug report assignment," in *Proc. 28th Int. Conf. Software Engineering (ICSE)*, 2006, pp. 937–940.

[5] Naguib *et al.*, "Bug report assignee recommendation using activity profiles," in *Proc. 10th Working Conf. Mining Software Repositories (MSR)*, 2013, pp. 22–30.

[6] Alazzam, A. AlEroud, Z. A. Latifah, and G. Karabatis, "Automatic bug triage in software systems using graph neighborhood relations for feature augmentation," *IEEE Trans. Computational Social Systems*, vol. 7, no. 5, pp. 1288–1303, 2020.

[7] G. Murphy and D. Cubranic, "Automatic bug triage using text categorization," in *Proc. 16th Int. Conf. Software Engineering and Knowledge Engineering (SEKE)*, 2004, pp. 1–6.

[8] Anvik, L. Hiew, and G. Murphy, "Who should fix this bug?" in *Proc. 28th Int. Conf. Software Engineering (ICSE)*, 2006, pp. 361–370.

[9] Xuan, H. Jiang, Y. Hu, Z. Ren, W. Zou, Z. Luo, and X. Wu, "Towards effective bug triage with software data reduction techniques," *IEEE Trans. Knowledge and Data Engineering*, vol. 27, no. 1, pp. 264–280, 2015.

[10] X. Xia *et al.*, "Improving automated bug triaging with specialized topic model," *IEEE Trans. Software Engineering*, vol. 43, no. 3, pp. 272–297, 2016.

[11] S. Mani, A. Sankaran, and R. Aralikatte, "DeepTriage: Exploring the effectiveness of deep learning for bug triaging," in *Proc. ACM India Joint Int. Conf. Data Science and Management of Data*, 2019, pp. 171–179.

[12] S. Xi, Y. Yao, X. Xiao, F. Xu, and J. Lu, "Bug triaging based on tossing sequence modeling," *J. Computer Science and Technology*, vol. 34, no. 5, pp. 942–956, 2019.

[13] A. Choquette-Choo *et al.*, "A multi-label, dual-output deep neural network for automated bug triaging," in *Proc. IEEE Int. Conf. Machine Learning and Applications (ICMLA)*, 2019, pp. 937–944.

[14] R. Wang, X. Ji, S. Xu, Y. Tian, S. Jiang, and R. Huang, "An empirical assessment of different word embedding and deep learning models for bug assignment," *J. Systems and Software*, vol. 210, p. 111961, 2024.

[15] S. F. A. Zaidi and C. G. Lee, "Learning graph representation of bug reports to triage bugs using graph convolution network," in *Proc. Int. Conf. Information Networking (ICOIN)*, 2021, pp. 504–507.

[16] Y. X. Li *et al.*, "A graph convolutional neural network-based approach for software bug triage,"

J. Wuhan Univ. (Natural Science Edition), vol. 66, no. 3, pp. 244–252, 2020.

[17] J. Dai, Q. Li, H. Xue, Z. Luo, Y. Wang, and S. Zhan, "Graph collaborative filtering-based bug triaging," *J. Systems and Software*, vol. 200, p. 111667, 2023.

[18] 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.

[19] H. Cao and M. Cui, "Complex network neural dynamics framework for automated software bug triaging," *IEEE Access*, 2025, doi: 10.1109/ACCESS.2025.3553819.