Graph Neural Network vs. Large Language Model: A Comparative Analysis for Bug Report Priority and Severity Prediction

Jagrit Acharya
University of Calgary
Calgary, Canada
jagrit.acharya1@ucalgary.ca

Gouri Ginde
University of Calgary
Calgary, Canada
gouri.deshpande@ucalgary.ca

ABSTRACT

A vast number of incoming bug reports demand effective methods to identify priority and severity for bug triaging. With increased technological advancement, machine learning and deep learning have been extensively examined to address this problem. Although Large Language Models (LLMs) such as Fine-tuned BERT (early generation LLM) have proven to capture context in the underlying textual data, severity and priority prediction demand additional features for understanding the relationships with other bug reports. This work utilizes the graph-based approach to model the bug reports and their other attributes, such as component, product and bug type information. It utilizes the relational intelligence of Graph Neural Network (GNN) to address the prioritization and severity assessment of bug reports in the Bugzilla bug tracking system. Initial tests on the Mozilla project dataset indicate that a project-wise predictive approach using GNNs yields higher accuracy in determining the priority and severity of bug reports compared to LLMs across multiple Mozilla projects, contributing to a notable advancement in the automation of bug severity and priority prediction tasks. Specifically, GNNs demonstrated a remarkable improvement over LLMs, increasing the priority prediction accuracy by 37% & 30% and severity prediction accuracy by 43% & 30% for Core and Firefox projects, respectively. Overall, GNN outperformed the Fine-tuned BERT (LLM) in predicting priority and severity for all the Mozilla projects.

CCS CONCEPTS

• **Software and its engineering → Software creation and management**; *Software verification and validation*; *Empirical software validation*;

KEYWORDS

Requirement Engineering, Large Language Model, BERT, Graph Neural Networks, Natural Language Processing

ACM Reference Format:
Jagrit Acharya and Gouri Ginde. 2024. Graph Neural Network vs. Large Language Model: A Comparative Analysis for Bug Report Priority and Severity Prediction. In *Proceedings of the 20th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE '24), July 16, 2024, Porto de Galinhas, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3663533.3664042

1 INTRODUCTION

Bug priority prediction helps software developers focus their efforts on the most critical bugs affecting the software's core functionality. Bug prioritization is essential for effective resource allocation and planning for fixing activities and additional testing [25]. Open source bug tracking tools such as Bugzilla [4] facilitate such platforms to manage the bug reports effectively. However, due to a vast number of incoming bug reports, engineering managers and analysts responsible for bug triaging [8] grapple with effective methods to identify the priority and severity of these issue reports at the project level. Automated priority and severity prediction has been explored in the past [25] [26] [35].

Chen et al. [7] uses deep learning to automate parts of incident management by prioritizing alerts based on historical data empirically. This method focuses on the most crucial alerts and helps manage the high volume of bug reports thereby improving bug triaging processes. However, identifying the effective method at a fine granular level remains an open question. We explore how bug reports can be represented holistically utilizing other aspects and attributes such as product type, component and bug type, which can capture important and useful information for further automation using prediction models.

Need for graph-based analysis: In Bugzilla, bug reports' severity value describes the impact of a bug report. This field is utilized by the programmers/engineers/release managers/managers to determine the severity and used as input for the priority of the bug report. It is recommended that it should not be set by people filing bugs. The severity value determines how developers working on the bug reports can effectively organize their work and further prioritize (set and reset new priority value) their issues. Thus, determining the severity of bug reports is the first step of issue triaging and has a wider impact on release planning and resource utilization. The severity value for the Defect bug report cannot be null, unlike Enhancement and Task bug reports [42]. Thus, other properties of the bug report, such as bug report type in addition to component, product and summary, are crucial to determine the severity and priority values of the bug reports.

Graph convolutional networks (GCNs) are a subset of graph neural networks (GNNs) that have demonstrated strong performance classification tasks. Researchers have begun extending these methods to heterogeneous graphs, with Yao et al. [44] developing a notable GCN-based classification approach. Graph Neural

Networks (GNNs) have emerged as a powerful tool in software engineering, especially in the context of bug report analysis. Zhou et al. [45] highlight the unique capability of GNNs to capture complex relationships and structural information within data, a feature that is particularly beneficial for assessing the severity and priority of bug reports in software projects. In contrast, while Large Language Models (LLMs) such as fine-tuned BERT have shown proficiency in textual content analysis, they may not fully exploit the intricate web of relationships characterizing bug reports. These relationships include connections among various bugs and their attributes like type, component, and product. These are crucial for a nuanced understanding of each bug's implications within the software's ecosystem.

In this study, we use attributes such as summary, product, component, and bug report type to capture a holistic view of the bug report and generate a graph-based representation of the data. Such modelled data is further processed using Graph Neural Network (GNN) algorithms. Its results are compared with the Large Language Model, specifically a fine-tuned Bidirectional Encoder Representations from the Transformers (BERT) model, which serves as our baseline. Thus, we explore two research questions (RQs) as follows.

RQ1 How do Language Models (LLM) and Graph Neural Networks (GNNs) compare in terms of accuracy in predicting the severity of issues within the Mozilla family of projects?

RQ2 How do Language Models (LLMs) and Graph Neural Networks (GNNs) compare in terms of accuracy in predicting the priority of issues within the Mozilla family of projects?

Our contributions are as follows:

- We propose a novel graph-based dataset representation to capture holistic bug report information. This representation will leverage GraphSAGE and Graph Attention Networks (GAT) to effectively model the complex relationships within the data. To establish meaningful connections among graph nodes, we'll employ cosine similarity to determine their relationship. This approach aims to consolidate information and enhance the modelling of bug report data.
- We explore the utility of this graph-based representation for bug report severity and priority prediction at a project level (fine-granular level) for the projects: Core, Firefox, DevTools and Testing from the Mozilla family of projects using GNN.
- We compare our method with the LLM (fine-tuned BERT) model and provide evidence for effective representation of information and knowledge using undersampling and oversampling () data balancing techniques.
- Our fine granular analysis of the project-level information for the Mozilla family of products relies on the approximately 0.5 million bug reports spanning from 2015 to 2023 we gathered from the online Bugzilla repository.

The structure of the rest of the paper is as follows. Section 2 details existing literature, and Section 3 provides information about the data collection and pre-processing. In Section 4 we elaborate on the methodology and study design followed by results in Section 5. Section 6 lists various limitations and Section 7 explains conclusion and future work.

2 RELATED WORK

Bug report priority and severity have been extensively analyzed and explored in the Software Engineering research community. This section discusses and elaborates on existing techniques and methods utilized in the literature to predict software bug severity and priority.

2.1 Conventional Machine Learning-Based Approaches

Cubranic and Murphy et al. [26] addressed the challenge of task prioritization through the lens of Machine Learning (ML), framing it as a classification problem in their research. Various other researchers have recently analyzed alternative methodologies and strategies to enhance the efficiency of bug-fix prioritization and assignment processes.

Menzies and Marcus [25] pioneered a distinct conceptual framework to automate the severity prediction of bug reports. They proposed the SEVERity Issue Assessment (SEVERIS) method, which leverages a machine learning classifier to predict the severity of bug reports by examining feature vectors built from the top k feature words. This contribution significantly enriched subsequent research in bug priority prediction, including studies by [31], [34]. Following up on this work, [17] [22] extended its application to the domain of open-source software bug repositories. Tan et al.[33] introduced an innovative approach for assessing the severity of bug reports by associating them with corresponding entries in the Stack Overflow bug repository. They employed three distinct classification algorithms for their study: K-nearest neighbours (KNN), Naive Bayes, and Long Short-Term Memory (LSTM) networks and significantly enhanced the mean F1-score.

2.2 Deep Learning-Based Approaches

The field of artificial intelligence has witnessed a significant evolution with the advent of Large Language Models (LLMs) like Transformers, BERT, and GPT [38][18]. Renowned for their versatility, these models have set new benchmarks across various Natural Language Processing (NLP) task applications. These models have been predominantly successful in capturing and interpreting contextual information from textual data [2][9].

A recent study by Ali et al. [1] introduced an approach where a fine-tuned BERT model was applied to mobile application reviews to predict severity classes. The dataset employed in their study was composed exclusively of app reviews alongside associated dates and severity ratings, resulting in a predominantly text-based dataset. Their methodology demonstrated superior performance over several deep learning models, showcasing the efficacy of fine-tuned BERT models in extracting meaningful insights from text-heavy data for severity prediction tasks.

Graph Neural Networks (GNN) have been extensively studied in understanding and interpreting graph-structured data. Their capability to process and analyze graph data through recursive message passing and aggregation mechanisms has been fundamental in various applications, including software project management [21, 44]. Recent work by Dong et al. [10] applied a GNN framework to determine which developer should fix a bug, enhancing prediction accuracy by 15% for specific projects as compared to previous work.

This improvement highlights GNN's effectiveness in bug-related prediction tasks. Panda et al. [28] used fuzzy similarity to propose bug priority and achieved promising results.

2.3 Applications of GNN

GNNs have demonstrated significant success in multiple areas, including NLP tasks [43], especially in recommender systems [13], in healthcare analytics [29], and other diverse applications [20]. Taking inspiration from this scholarly work, in this research, we propose to interpret the bug report and its attributes in a graph-based structure to further utilize GNN for priority and severity prediction.

3 DATA COLLECTION

In this study, we developed a Python script to systematically collect and refine data from the Bugzilla bug tracking platform [4]. We gathered 452,257 bug reports from June 2015 through December 2023 using Bugzilla's API library calls [3] iterating through bug IDs.

Bug reports in Bugzilla can be one of the three types: Enhancements, Defects and Tasks. Each bug report returned by the API had several attributes, and we identified 11 useful ones for our study. More details about the selected attributes (a.k.a properties) are in Table 1. Attribute, *ID*, serves as a unique identifier for each bug report and is used to uniquely identify every node in our graph representation Figure 4. *Summary* provides a brief description of the bug report in a string format, allowing for a quick overview of the problem at hand; it is used to draw edges between the nodes for the graph. The *Product* attribute indicates the product or project with which the issue is associated, with the dataset covering four Mozilla products: Core, Firefox, DevTools, and Testing. The *Component* attribute refers to the specific part of the product that is affected by the issue which is used a the node feature of the graph's node. The *Blocks* attribute indicates other issues that are blocked by this current bug. In contrast, the *Depend_on* attribute identifies other bugs that the current bug relies on or is dependent upon. Additionally, the *Status* attribute provides information about the current state of the bug report, such as whether it is assigned, unassigned, resolved, new, etc.

The *Priority* denotes the importance or urgency of addressing the bug, classified into five categories from P1 to P5, indicating the sequence in which issues should be tackled here: P1 is the highest Priority, and P5 is the lowest. Lastly, for *Severity*, set by the person raising the bug report, it helps programmers/engineers/release managers prioritize the bug report further. The dataset was consolidated into four principal classes: Normal, S3, S4, and Critical. This consolidation was due to the relatively low incidence of issues in the Major, S2, Minor, Trivial, Blocker, Enhancement, and S1 categories, which rendered them less impactful for the ML training process.

Figure 1 shows the data distribution for the four projects based on various *Severity* categories. The Core and Firefox project have large number of samples from each severity type. Hence, only normal, S3, S4 and critical classes are selected as they have relatively larger samples for each of the four projects (Core, Devtools, Firefox and Testing).

Figure 2 shows the data distribution for the four projects based on various *Priority* categories. All five priority classes have a large number of samples for Core and Firefox projects compared to Testing and DevTools. For the DevTools project, Class P4 samples are fever in number. We have made the complete dataset and source code open source[1]

4 METHODOLOGY AND ARCHITECTURE

Figure 3 shows the high level architecture of our study. The dataset is mined using Buzilla API. Other steps of the study design are explained as follows.

4.1 Data Pre-processing

For our study, after several experiments and feature importance tests, we determined that the attributes *Blocks*, *Depends_on*, and *Status* did not yield and negatively impacted the predictive results. We utilized a systematic evaluation employing several statistical tests and analytical techniques, including Random Forest Feature Importance, Pearson Correlation Coefficients, and Multicollinearity Assessment via Variance Inflation Factor (VIF) analysis to determine their exclusion as implemented by researchers [23, 24, 30].

Further, this raw dataset was passed through several stages of the preprocessing pipeline. We chose the top 4 projects out of the 150 projects in our dataset since these top projects contributed to over 50% of the total data points as shown in distribution charts in Figure 2 and Figure 1 for priority and severity types respectively. Various preprocessing steps carried out are as follows:

Text normalization: All *Summary* text data was converted to lowercase to maintain uniformity and facilitate easier analysis.

Punctuation and stopword removal: We eliminated punctuation marks, special characters, and common English stopwords from the text. This step was aimed at emphasizing the meaningful content within the dataset.

Null value elimination: We removed all entries with null or missing values in the priority and severity attributes to ensure data integrity. This removed 105,515 data points with one or more empty values for the data sample set focusing on the priority dataset and 25,968 rows from the severity-focusing dataset.

Categorical to numerical conversion: We converted categorical data present in the priority and severity attributes into a numerical format using label encoding. This transformation facilitates applying various data analysis and ML algorithms that require numerical input.

Filtering: Data points featuring *Summary* attribute with fewer than five words were excluded from the dataset, totalling 48,284 instances.

4.2 Graph Structure

For our proposed approach of modelling the dataset using a graph network, it has nodes, edges, node properties and edge properties as shown in Figure 4. We convert summary attribute content into Term Frequency-Inverse Document Frequency (TF-IDF) vectors, adopting a technique previously utilized for severity and priority prediction [28, 32]. These vectors, together with ID, one-hot encoded representations of component and bug report type, and

[1]https://doi.org/10.5281/zenodo.10892319

Table 1: Content and structural information about the dataset

Property	Description	Value
ID	Unique identifier	Number
Summary	A brief description of the bug report	String
Product	The product or project the issue is associated with	Four products [41]: Core - For bugs in the shared components used by Firefox and other Mozilla software, including handling of Web content, Firefox - An open-source web browser, DevTools - A set of web developer tools built into Firefox, Testing - For bug reports related to automated testing of Mozilla client code (Firefox, Thunderbird, Fennec, Gecko, etc.)
Component	A specific part of the product affected by the issue	Framework, Debugger, Console, General
Bug types/ Issue type	The nature or category of the reported bug	Three types: Task (refactoring, removal, replacement, enabling or disabling of functionality and any other engineering task), Defect (regression, crash, hang, security vulnerability and any other general issue), Enhancement (new feature, improvement in UI etc.)
Priority	The importance or urgency of fixing the issue	Five Priority categories: P1 (Fix in the current release cycle), P2 (Fix in the next release cycle or the following (nightly + 1 or nightly + 2), P3 (Backlog), P4 (Do not use, this priority is for web platform test bots), P5 (Will not fix) and – (No decision) [12]
Severity	The impact level of the issue on the product	Four Severity categories: Normal, S3 (Blocks non-critical functionality), S4 (low or no impact to users), Critical (Blocks development/testing) [42]

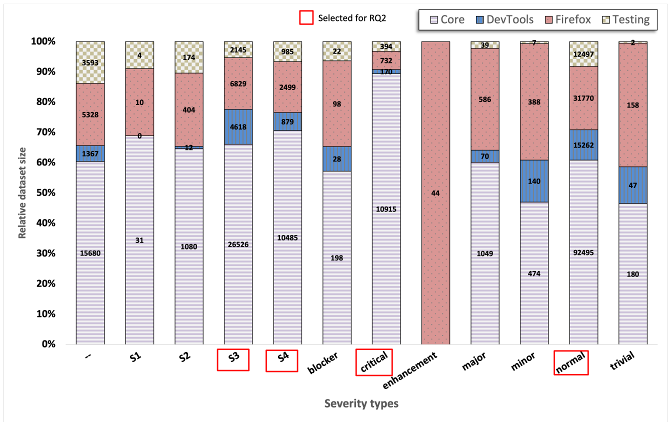

Figure 1: Data distribution based on various severity types: Core and Firefox projects have a large number of samples from each severity type. Only *normal, S3, S4 and critical* classes are selected as they have relatively larger samples for each one of the projects (Core, Devtools, Firefox and Testing).

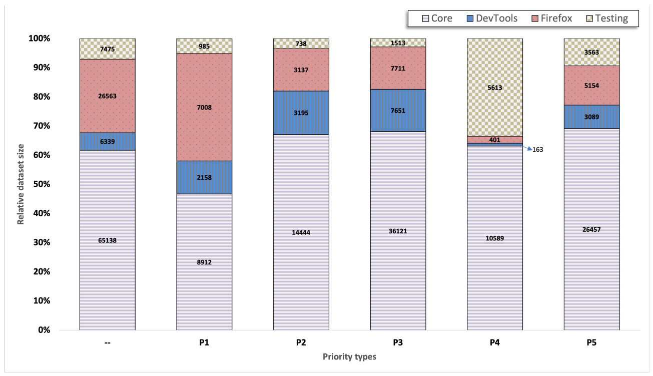

Figure 2: Data distribution based on various priority types: All five priority classes have a large number of samples for Core and Firefox projects compared to Testing and DevTools. In general, the class imbalance is evident in this data distribution chart

severity (when predicting priority) or priority (when predicting severity), are incorporated as node features. The edges within the graph are established by calculating the cosine similarity between the TF-IDF vectors of the summaries. We determined 0.95 as the cosine similarity threshold for creating the edge between any two vector representations based on various experiments with varying threshold values. Experiments showed that lower threshold values significantly increased the degree of the nodes by doubling or even tripling it, and this resulted in a substantially denser graph and lowered the evaluation scores, a finding that aligns closely with those reported in [40].

4.3 Graph Neural Network

In our study of graph neural networks, we tested various architectures, including Graph Convolutional Network (GCN), Graph Attention Network (GAT), and GraphSAGE. We evaluated these

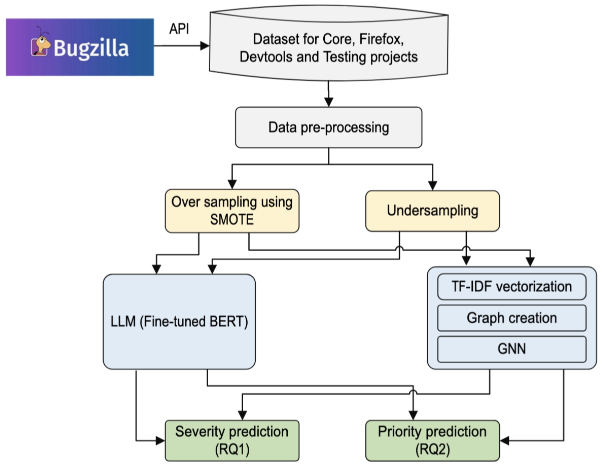
Bugzilla
API
Dataset for Core, Firefox, Devtools and Testing projects
Data pre-processing
Over sampling using SMOTE
Undersampling
LLM (Fine-tuned BERT)
TF-IDF vectorization
Graph creation
GNN
Severity prediction (RQ1)
Priority prediction (RQ2)

Figure 3: Overall research design

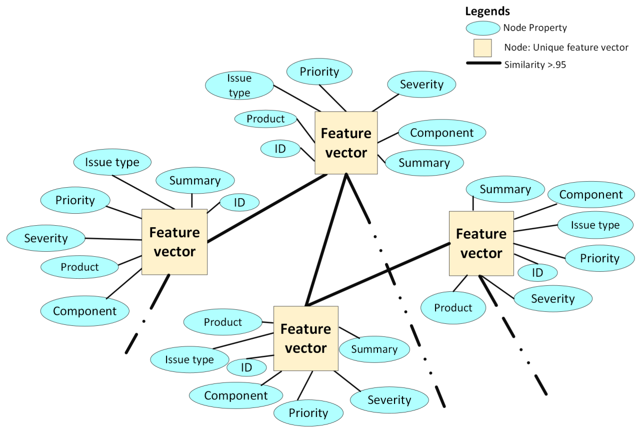

Figure 4: Graph network design depicting various nodes, properties, and edges for the bug reports dataset. In this model, if priority is the data label, severity is used in the feature vector, and if severity is the data label, priority is used.

models based on their capacity to learn graph-structured data with node features and graph topology. Our finding shows that GNN model with GraphSAGE layers, followed by GAT layers delivered superior performance for multi-class classification.

Our graph $G = (V, E)$, where V represents the set of nodes (issues) and E represents the set of edges (relationships between issues based on similarity of their descriptions). Each node $v \in V$ is associated with a feature vector $\mathbf{x}_v$ derived from a combination of TF-IDF and one-hot encoded categorical attributes.

The GNN model architecture is initiated with two main parameters: the number of node features, which represents the number of features each node has, and the number of classes, which denotes the number of possible output classes for the classification task. Within the model, two types of convolutional layers are employed: GraphSAGE [15] and GAT [39], both of which are critical components of the graph neural network (GNN) approach.

The GraphSAGE (Graph Sample and Aggregation) layer updates node features by aggregating features of its immediate neighbours followed by a transformation via a weight matrix and a non-linear activation and the formula defined by Fey et al. [11]:

$$\mathbf{h}_v^{(k)} = \sigma\left(\mathbf{W}^{(k)} \cdot \text{MEAN}\left(\mathbf{h}_v^{(k-1)}, \left\{\mathbf{h}_u^{(k-1)} : u \in \mathcal{N}(v)\right\}\right)\right)$$

where:

- $\mathbf{h}_v^{(k)}$ denotes the feature vector of node v at the k-th layer.
- $\mathbf{W}^{(k)}$ is the layer-specific trainable weight matrix.
- σ is a non-linear activation function, in our case ReLU.
- $\mathcal{N}(v)$ denotes the set of neighbors of v in the graph G.
- MEAN represents the mean pooling aggregation function, computing the average of the central node's features and its neighbours' features at the previous layer.

The Graph Attention Network (GAT) introduces an attention mechanism that assigns different weights to different nodes in a neighborhood, allowing for more nuanced feature aggregation: The input to our GAT layer is a set of node features, $\mathbf{h} = \{\vec{h}_1, \vec{h}_2, \ldots, \vec{h}_N\}$, where $\vec{h}_i \in \mathbb{R}^F$, and N is the number of nodes, and F is the number of features in each node.

According to Fey et. al [11] we use the following equation to get attention scores

$$\alpha_{ij} = \frac{\exp(\text{LeakyReLU}(a^T[\mathbf{W}\mathbf{h}_i \| \mathbf{W}\mathbf{h}_j]))}{\sum_{k \in \mathcal{N}(i)} \exp(\text{LeakyReLU}(a^T[\mathbf{W}\mathbf{h}_i \| \mathbf{W}\mathbf{h}_k]))}$$

$$\mathbf{h}_i' = \sigma\left(\sum_{j \in \mathcal{N}(i)} \alpha_{ij} \mathbf{W}\mathbf{h}_j\right)$$

where:

- α_{ij} is the attention coefficient computed between node i and node j, indicating the importance of node j's features to node i.
- a and $\mathbf{W}$ are learnable parameters of the model, with a being a vector that projects the concatenated features into a scalar and $\mathbf{W}$ being the shared linear transformation applied to the features before attention coefficients are computed.
- $\|$ denotes concatenation of feature vectors.
- The function LeakyReLU is used to introduce non-linearity in the computation of attention coefficients.
- σ is a non-linear activation function, and $\mathbf{h}_i'$ represents the updated feature vector of node i.

The model's output is subjected to a log softmax operation, which is suitable for multi-class classification tasks because it is a variant of the softmax function that converts the raw output scores from the network and normalizes these values. The softmax output indicates the probability of each class for a given bug. The class with the highest probability is chosen as the final classification. This structure enables the model to effectively learn from the graph-structured data, leveraging the node features and the graph topology. The model, alongside an optimizer and a loss function, is also initialized to facilitate the training process. This comprehensive setup forms the basis for training the GNN model to accurately predict issue priority and severity further.

4.4 Fine-Tuned BERT (LLM)

LLM model such as Bidirectional Encoder Representations from Transformers (BERT) is a pre-trained transformer model known for its effectiveness in various natural language processing (NLP) tasks and text classification. Also, it is widely used in Software Engineering tasks such as bug report priority and severity prediction. In this study, we utilize fine-tuned BERT [9] as our baseline to predict the priority and severity of bug reports. Thus, we fine-tuned the *bert-base-uncased* variant of the BERT model for the classification task, integrating it with additional categorical data derived from our dataset. This dataset was preprocessed, generating BERT embedding of text summaries and one-hot encoding of categorical features for compatibility with our model architecture, which incorporates a custom *BertClassifier*. This classifier extends the original BERT structure with a dropout (ratio = 0.1) and a linear classification layer, processing textual and categorical inputs. The model was optimized over three epochs using the Adam optimizer with a 2e-5 learning rate and a batch size 16.

Table 2: Train, test, and validation sample size for Priority prediction model training

Project	Train set	Test set	Validation set	#Classes
Core	27,000	9,000	9,000	5
Firefox	11,117	3,706	3,706	5
Devtools	5,277	1,759	1,759	5
Testing	2,305	768	768	5

4.5 Managing Skewed Dataset

Figure 1 shows that the dataset exhibits a prevalence of entries with the *Normal* severity label over other categories, while entries tagged with the *Enhancement* severity label are considerably the least. Similarly, within the spectrum of priority labels, Figure 2 shows that the P3 type outnumbered its other priority types for almost all the Mozilla products examined.

To effectively balance the dataset, we used two techniques, i.e. undersampling and the Synthetic Minority Over-sampling Technique (SMOTE) techniques in order to create two different datasets to test the models. Initially, undersampling was applied to reduce the size of the majority class, bringing its quantity closer to that of the minority class. Following this, we created a new dataset using SMOTE to further augment the minority class by synthesizing new examples, thereby ensuring an equal number of observations for each class in both datasets used for training and comparing our models. we test our proposed method (GNN) and the baseline (LLM) using two sampling techniques:

- Undersampling: This technique reduces the size of the majority class to match the minority class, thereby balancing the dataset.
- Oversampling: The Synthetic Minority Over-sampling Technique (SMOTE) [6] is a method widely used for its ability to generate synthetic instances based on feature space similarities within the minority class, rather than merely replicating existing data. This approach has effectively demonstrated mitigating the issues associated with traditional oversampling and under-sampling methods, thereby ensuring a more equitable distribution of data across classes [14][16]

.

Table 3: Train and test and validation set size information for Severity prediction model training

Project	Train	Test	Validation	#Classes
Core	27,000	9,000	9,000	4
Firefox	6,739	2,246	2,246	4
Devtools*	1,752	584	584	3
Testing	2,009	670	670	4

* Due to lack of enough train set for *Critical* class, we exclude it for Severity analysis

4.6 Model Training

To train the ML models, we split the dataset into distinct subsets for training, validation, and testing in the ratio of 60%, 20%, and 20%, respectively. The validation test was specifically used for hyperparameter tuning. This partitioning approach helps in tuning the initial parameters of the model effectively. Additionally, we incorporate a 10-fold cross-validation method to ensure a thorough evaluation and generalizability of our model across accuracy, precision, recall, and F1-score, enhancing its reliability and effectiveness in handling varied data distributions. Table 3 and 2 show statistical details about the Train, test, and validation set sizes used for this study.

4.7 Computational Resources

Our models, including the GNN architecture and the fine-tuned BERT, were trained on NVIDIA Tesla A100 GPUs with 50 GB of RAM and a 30-core CPU setup. These GPUs provided the necessary computational power to rapidly train and handle our extensive datasets and complex model architectures. The efficiency and scalability of our experiments were significantly enhanced by these resources, ensuring the reproducibility of our study in comparable high-performance computing environments.

4.8 Model Evaluation

Four key measures are pivotal in evaluating the proposed models for priority and severity classification tasks: accuracy, recall, precision, and the F1 score, as consistently highlighted in the priority and severity prediction research [37]. We evaluated LLM and GNN models in terms of these four measures for category (i) as below.

$$Precision(i) = \frac{TP(i)}{TP(i) + FP(i)} \tag{1}$$

$$Recall(i) = \frac{TP(i)}{TP(i) + FN(i)} \tag{2}$$

$$F1(i) = \frac{2 \times Precision(i)Recall(i)}{Precision(i) + Recall(i)} \tag{3}$$

$$Accuracy(i) = \frac{TP(i) + TN(i)}{TP(i) + TN(i) + FP(i) + FN(i)} \tag{4}$$

$TP(i)$: Bug report truly classified as category i
$FP(i)$: Bug report falsely classified as category i
$FN(i)$: Bug report falsely classified as not being category i
$TN(i)$: Bug report truly classified as not being category i

5 RESULTS

The research questions are evaluated using two methods, LLM and GNN, and two sampling techniques, Undersampling and Oversampling (using SMOTE). This section presents the results of these experiments.

5.1 Answering RQ1: Bug Report Severity Prediction

For severity prediction, we used to oversample data samples across all severity labels of Mozilla Projects, setting a balanced stage for GNN's application. The effects of applying GNN and LLM to different projects are illustrated in Figure 6. At the same time, the outcomes with undersampling are depicted in Figure 5.

5.1.1 *Undersampling:*

Figure 5 shows the undersampling results through the GNN and LLM performance metrics.

- GNN achieved a 0.96 across accuracy, precision, recall, and F1 score for the Mozilla Core project. In contrast, LLM scored significantly lower scores of accuracy = 0.67 and 0.55 for precision and recall. For the Mozilla Core project, the GNN model consistently achieved a score of 0.96 in accuracy, precision, recall, and F1 score. In contrast, the BERT model demonstrated significantly lower performance, with an accuracy of 0.67 and a precision and recall score of 0.55. Thus, increasing the overall priority prediction by 43%.
- Within the Core Project, GNN's uniform dominance over LLM with a decent margin because of the sufficient size of training data for all the classes and effectively capture relationships and structures, which can also be evident in Table 4 where GNN was particularly effective across all classes in the Core Project, showing consistently good performance.
- In contrast, the LLM showed uneven performance; for instance, it did better in the *Normal* class than in others. The Firefox project shows GNN maintaining decent performance at 0.78 in all categories, outperforming BERT's 0.60-0.62 range.
- In the Testing project, GNN recorded a 0.82 in accuracy and recall, with LLM lagging at 0.80 for accuracy and lower in other measures. We attribute this to skewed class distribution compared to other three projects.
- The DevTools project further confirmed GNN's efficacy, with scores of 0.69 in accuracy and recall, overshadowing LLM's 0.6 in accuracy and around 0.55 in other metrics.

GNN consistently surpasses the performance of LLM across all projects, especially for projects with imbalanced datasets. Utilizing GNN, the overall measure scores increased by 10-20% (comparing Figure 5 and Figure 6). For the Core project, the measure scores remain constant due to a relatively balanced dataset for all the classes, while the projects with an imbalance dataset have an increase of up to 20%

5.1.2 *Oversampling:*

Figure 6 shows the severity prediction with SMOTE. The comparative analysis of the model performances across various projects (after balancing the data, Core and Firefox already have a balanced set of data, hence, no improvement was shown there) showed improved performance for the Testing and DevTools dataset. Contrary to the Core and Firefox datasets, these projects had a significant imbalanced data. Thus, utilizing SMOTE demonstrated relatively good improvement across all the classes and evaluation measures.

- GNN consistently demonstrated superior predictive capabilities over the BERT, as evident in their accuracy, precision, recall, and F1 score metrics. Specifically, within the Core Project, GNN showed uniform scores of 0.96, significantly ahead of LLM's consistent 0.67 across all metrics, highlighting GNN's robustness in complex data environments.
- The Firefox project showed a narrower performance gap, with GNN at 0.76 and LLM at 0.61, indicating a competitive edge for GNN.
- Interestingly, the Testing project revealed a scenario where LLM slightly edged out GNN in accuracy (0.82 vs. 0.81) and showcased close matches in other metrics, suggesting LLM's nuanced effectiveness in certain conditions.
- For DevTools, both models performed commendably, with GNN maintaining a high consistency at 0.81 across metrics, compared to LLM's substantial improvement to 0.70, underscoring GNN's overall superiority while acknowledging LLM's adaptability and potential under specific scenarios.

For *Severity* prediction, using the oversampling technique, both GNN and LLM showed comparable performance, with GNN leading in overall measures. However, LLM demonstrates notable strengths in a few scenarios.

5.2 Answering RQ2: Bug Report Priority Prediction

Similar to RQ1, we utilize two methods and two sampling techniques to predict the priority of bug reports and compare the outcome based on various measures. Figure 8 shows the comparative results using oversampling, while the outcomes with undersampling are shown in Figure 7

For *Priority* prediction, the GNN consistently surpasses the performance of LLM across all projects. Notably, for projects with imbalanced datasets, GNN outperforms by 8-10% in all evaluation metrics. However there are no significant changes in scores for Core and Firefox projects, as the mined dataset was already relatively balanced (Figure 2).

The Core Project had a relatively large and balanced dataset, which reflected well in performance measures where accuracy, F1 score, recall, and precision reached 0.92. This also emphasizes that GNN performs well with large and balanced datasets.

5.2.1 *Undersampling:*

In our comparative analysis of GNN and LLM for predicting priority undersampling as shown in Figure 7.

- Across different projects, GNNs generally outperformed BERT (an early generation LLM) in the Core project across all the classes, which can be seen in Table 5, with significantly higher accuracy, precision, recall, and F1 scores (all metrics at 0.92 for GNN vs. 0.67 for LLM). Thus achieving an overall improvement of 37%.
- In the context of the Firefox and Testing projects, GNNs demonstrated their superiority over LLMs in most measures. However, the margin was narrow for the DevTools project, suggesting that more data can enhance the learning capabilities of both models. This nuanced performance highlights the unique strengths of GNNs and LLMs, with GNNs showing a better fit for tasks involving relational data.
- In contrast, LLMs cannot capture the relational dependencies. Still, they could offer competitive advantages in scenarios requiring advanced natural language understanding.

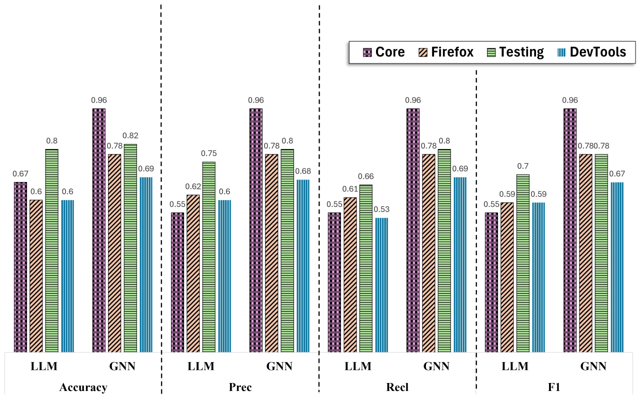

Figure 5: Bug severity prediction metrics: Accuracy, precision, recall and F1-score for 10-cross fold validation using undersampling for GNN and fine-tuned BERT without SMOTE

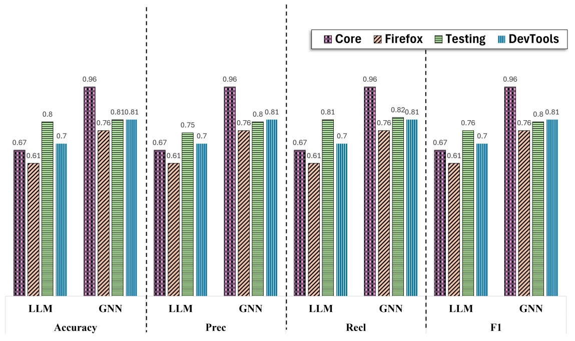

Figure 6: Bug severity prediction metrics: Accuracy, precision, recall and F1-score for 10-cross fold validation with for GNN and fine-tuned BERT wit SMOTE

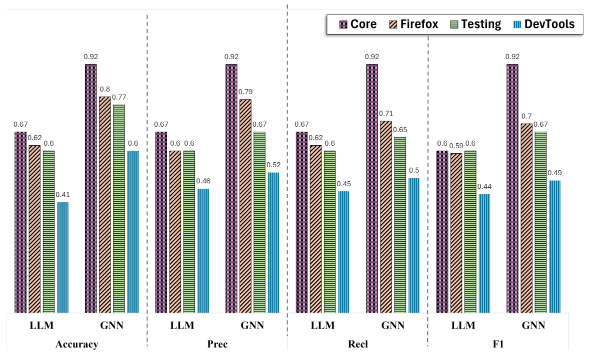

Figure 7: Bug priority prediction metrics: Accuracy, precision, recall and F1-score for 10-cross fold validation using undersampling for GNN and fine-tuned BERT without SMOTE

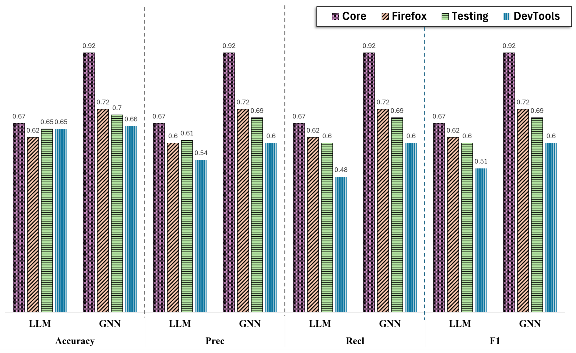

Figure 8: Bug priority prediction metrics: Accuracy, precision, recall and F1-score for 10-cross fold validation with for GNN and fine-tuned BERT with SMOTE

Table 4: Precision, Recall and F1-score values for various classes of Priority and Severity classes for the two larget projects: Core and Firefox

		Core (Undersampling)						Core (With SMOTE)						Firefox (Undersampling)						Firefox (With SMOTE)					
		LLM			GNN			LLM			GNN			LLM			GNN			LLM			GNN		
		Prec	Rec	F1	Prec	Rec	F1	Prec	Rec	F1	Prec	Rec	F1	Prec	Rec	F1	Prec	Rec	F1	Prec	Rec	F1	Prec	Rec	F1
Severity	Critical	0.64	0.65	0.66	**0.95**	**0.94**	**0.94**	0.64	0.66	0.65	0.95	0.93	0.95	0.54	0.5	0.49	0.66	0.44	0.53	0.67	0.54	0.58	0.68	0.43	0.58
	S3	0.46	0.45	0.46	**0.96**	0.96	0.95	0.47	0.45	0.44	0.96	0.97	0.96	0.5	0.63	0.57	0.79	0.74	0.76	0.55	0.67	0.62	0.78	0.73	0.75
	S4	0.72	0.66	0.61	0.95	0.95	0.94	0.73	0.65	0.62	0.96	0.95	0.97	0.69	0.66	0.7	0.91	**0.89**	**0.93**	0.72	0.57	0.65	0.93	0.88	**0.94**
	Normal	0.78	0.86	0.82	0.95	**0.97**	0.95	0.79	0.87	0.81	0.96	**0.97**	0.96	0.66	0.58	0.57	0.69	0.86	0.76	0.66	0.8	0.59	0.7	0.87	0.75
Priority	P1	0.6	0.42	0.52	0.94	0.92	**0.95**	0.59	0.51	0.53	0.9	**0.96**	0.91	0.38	0.36	0.37	0.7	0.9	0.79	0.45	0.41	0.42	0.7	0.9	0.79
	P2	0.42	0.54	0.47	0.89	0.9	0.9	0.45	0.58	0.46	0.9	0.88	0.89	0.4	0.32	0.31	0.65	0.45	0.25	0.5	0.36	0.35	0.66	0.16	0.25
	P3	0.44	0.47	0.45	0.9	0.89	0.87	0.45	0.46	0.45	0.94	0.86	0.9	0.45	0.49	0.43	**0.97**	0.96	**0.98**	0.55	0.49	0.49	0.97	**0.98**	**0.98**
	P4	0.97	0.85	0.91	0.94	0.92	0.93	0.96	0.96	0.9	0.95	0.92	0.94	0.1	0.05	0.12	0.11	0.04	0.09	0.29	0.25	0.27	0.3	0.31	0.34
	P5	0.71	0.84	0.77	0.94	0.95	0.91	0.76	0.88	0.78	0.97	0.96	0.94	0.69	0.73	0.7	0.82	0.82	0.82	0.76	0.76	0.75	0.81	0.82	0.82

- After an in-depth analysis of the graph, we found that the node features for the core graph are five times more features than those for other project graphs, which explains the richness of the content for the core project. The average degree of nodes in the core project graph is also double that of other project graphs.

Results show that with the undersampling technique, GNNs are better suited for tasks involving relational data, whereas LLMs struggle to capture relational dependencies. However, we speculate that the latest LLMs could show nuanced performance, and evaluation of which is part of our future work

Table 5: Precision, Recall and F1-score values for various classes of Priority and Severity classes for the two smaller projects: Testing and DevTools

		Testing (Undersampling)						Testing (With SMOTE)						DevTools (Undersampling)						DevTools (With SMOTE)					
		LLM			GNN			LLM			GNN			LLM			GNN			LLM			GNN		
		Prec	**Rec**	**F1**	**Prec**	**Rec**	**F1**	**Prec**	**Rec**	**F1**	**Prec**	**Rec**	**F1**	**Prec**	**Rec**	**F1**	**Prec**	**Rec**	**F1**	**Prec**	**Rec**	**F1**	**Prec**	**Rec**	**F1**
Severity	**Critical**	0.06	0.05	0.04	0.13	0.15	0.14	0.69	0.65	0.67	0.7	0.71	0.7												
	S3	0.72	0.65	0.74	0.81	0.7	0.84	0.75	0.75	0.8	0.86	0.82	0.85	0.47	0.4	0.45	0.68	0.6	0.63	0.73	0.7	0.72	0.82	0.72	0.77
	S4	0.74	0.5	0.53	0.76	0.75	0.58	0.76	0.64	0.73	0.79	0.75	0.72	0.54	0.5	0.53	0.69	0.67	0.68	0.75	0.82	0.75	0.8	0.83	0.79
	Normal	0.79	0.85	0.81	0.85	**0.95**	0.92	0.81	0.94	0.84	0.87	**0.95**	0.91	0.79	0.7	0.8	0.68	0.8	0.71	0.8	0.8	0.84	0.81	**0.9**	0.87
Priority	**P1**	0.52	0.63	0.55	0.62	0.55	0.6	0.5	0.61	0.53	0.58	0.55	0.55	0.4	0.41	0.38	0.43	0.53	0.62	0.43	0.53	0.62	0.63	0.61	0.6
	P2	0.43	0.35	0.36	0.3	0.21	0.26	0.43	0.42	0.38	0.48	0.47	0.46	0.42	0.32	0.35	0.4	0.58	0.37	0.4	0.58	0.37	0.54	0.44	0.56
	P3	0.55	0.5	0.55	0.68	0.58	0.56	0.63	0.5	0.56	0.69	0.58	0.56	0.6	0.64	0.65	0.66	0.64	0.68	0.66	0.64	0.68	0.76	0.96	0.75
	P4	0.67	0.74	0.77	0.89	**0.97**	0.92	0.74	0.79	0.78	0.9	**0.96**	0.94	0.06	0.1	0.06	0.08	0.14	0.09	0.08	0.14	0.09	0.18	0.15	0.2
	P5	0.72	0.73	0.74	0.9	0.96	0.92	0.75	0.73	0.76	0.9	0.96	0.94	0.81	0.8	0.79	0.85	0.81	0.82	0.85	0.81	0.82	0.86	0.85	0.88

The Core project of Firefox is the largest project (Figure 2) whose components are integral across all projects within the Mozilla ecosystem [5]. Thus, the classification scores are higher for the Core project than other projects considered for this study due to the meaningful patterns available for training models.

5.2.2 Oversampling: The figure 8 depicts that the measures for Core and Firefox projects remain unchanged because they already have a balanced dataset. Conversely, there is a significant difference in Devtools and Testing project results. We see an 11 % increase. This sharp enhancement underscores the efficacy of the algorithm in balancing the dataset, thereby significantly boosting the model's predictive performance.

6 THREATS TO VALIDITY

6.1 Internal Threats

The use of open-source software bug repository Bugzilla for Family of Mozilla projects in this research is characterized by their highly imbalanced data distribution. For higher severity and priority classes the data samples are marginally smaller compared to the other classes such as "Normal". To mitigate its impact while training we have used SMOTE oversampling method which relatively improved the results. Future studies could explore alternative data balancing methods on this repository for comparison. Furthermore, while we opted for the BERT (bert-uncased-model) , a early generation LLM model, for our analysis, the exploration of alternative models, such as ChatGPT [27] and Llama [36] may yield additional insights or improvements.

6.2 External Threats

This study analyzed bug reports from four open-source project bug repositories. The intrinsic quality of these projects plays a crucial role in determining the success of the analyzed techniques. It is important to note that commercial and industrial software projects, which adhere to distinct protocols compared to open-source projects, were not included in this analysis. To broaden the applicability of the findings, future research should consider applying the proposed methods to commercial and industrial software environments. Moreover, the research was based on severity and priority labels extracted from the repositories at a specific point in time. Given that these labels can be revised by moderators, even minor adjustments in the bug reports could potentially impact the effectiveness of the researched techniques.

6.3 Construct Threats

A possible concern for the validity of our construction lies in the selection of evaluation metrics. The metrics we chose, such as accuracy, precision, recall, and f-measure, are commonly used in studies of the similar nature.

7 CONCLUSION AND FUTURE WORK

Bug report severity and priority determination have broader resource and release management implications. Thus, determining the priority and severity of incoming new bug reports through machine learning-based automation has been of interest and has been explored intensively in the Software Engineering domain recently. Our study proposes graph-based representation considering various attributes of the bug report such as product type, component name, severity (while predicting priority), priority (while predicting severity), and bug type to represent the bug report holistically through nodes, edges and node properties. Various experiments and comparisons done using data mined from Bugzilla showed that Graph Neural Network (GNN) outperformed early-gneration LLM (fine-tuned BERT) for both priority and severity prediction by a substantial margin, emphasizing that considering other details of the bug report for training has significant implications on the overall machine learner's performance.

GraphSAGE scales to large graphs; however, it may require more epochs to train effectively. Whereas, GAT is moderately scalable and can capture fine-grained relationships effectively [19]. While our proposed GNN-based approach looks promising, its effectiveness when compared to state-of-the-art LLMs [27], [36] for large-scale software projects, remains part of our future work.

In future, we will explore real-world software projects to investigate the scalability of our approach. Our future explorations will focus on how BERT embeddings could capture hybrid information in a graph neural network. Additionally, we will identify transfer learning in this context to predict the severity and priority of bug reports belonging to a smaller project with a limited training dataset. Comparisons between GNN and state-of-the-art models such as ChatGPT [27] and Llama [36] will also be conducted to evaluate their respective efficacies for severity and priority prediction.

ACKNOWLEDGMENT

We thank Aakash Sorathiya and Jacob Idoko, Masters students at the Software Hub for AnalytiKs, Technology and Innovation lab, University of Calgary for reviewing the initial draft of the manuscript and providing critical comments. Also, this research was partially supported by the Natural Sciences and Engineering Research Council of Canada, NSERC Discovery Grant RGPIN-2023-03365.

REFERENCES

[1] Asif Ali, Yuanqing Xia, Qasim Umer, and Mohamed Osman. 2024. BERT based severity prediction of bug reports for the maintenance of mobile applications. *Journal of Systems and Software* 208 (2024), 111–898.

[2] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[3] Bugzilla. 2024. Bugzilla API. https://bugzilla.mozilla.org/rest/bug/1000. [Accessed 21-03-2024].

[4] Bugzilla. 2024. Bugzilla website — bugzilla.mozilla.org. https://bugzilla.mozilla.org/home. [Accessed 21-03-2024].

[5] Bugzilla. 2024. Components for Core. https://bugzilla.mozilla.org/describecomponents.cgi?product=Core. [Accessed 21-03-2024].

[6] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research* 16 (2002), 321–357.

[7] Junjie Chen, Shu Zhang, Xiaoting He, Lin, et al. 2021. How incidental are the incidents? characterizing and prioritizing incidents for large-scale online service systems. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 373–384. https://doi.org/10.1145/3324884.3416624

[8] Anh-Hien Dao and Cheng-Zen Yang. 2021. Severity prediction for bug reports using multi-aspect features: a deep learning approach. *Mathematics* 9, 14 (2021), 16–44.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[10] Haozhen Dong, Hongmin Ren, Jialiang Shi, Yichen Xie, and Xudong Hu. 2024. Neighborhood contrastive learning-based graph neural network for bug triaging. *Science of Computer Programming* 235 (2024), 103093. https://doi.org/10.1016/j.scico.2024.103093

[11] Matthias Fey and Jan Eric Lenssen. 2019. Fast graph representation learning with PyTorch Geometric. *arXiv preprint arXiv:1903.02428* (2019).

[12] Firefox. 2024. Priority Definitions; Firefox Source Docs documentation — firefox-source-docs.mozilla.org. https://firefox-source-docs.mozilla.org/bug-mgmt/guides/priority.html. [Accessed 23-03-2024].

[13] Chen Gao, Yu Zheng, Nian Li, Yinfeng Li, Qin, et al. 2023. A survey of graph neural networks for recommender systems: Challenges, methods, and directions. *ACM Transactions on Recommender Systems* 1, 1 (2023), 1–51.

[14] Abeer Hamdy and Abdulrahman El-Laithy. 2019. Smote and feature selection for more effective bug severity prediction. *International Journal of Software Engineering and Knowledge Engineering* 29, 06 (2019), 897–919.

[15] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).

[16] Haibo He and Edwardo A Garcia. 2009. Learning from imbalanced data. *IEEE Transactions on knowledge and data engineering* 21, 9 (2009), 1263–1284.

[17] Jerónimo Hernández-González, Daniel Rodriguez, Iñaki Inza, Rachel Harrison, and Jose A. Lozano. 2018. Learning to classify software defects from crowds: A novel approach. *Applied Soft Computing* 62 (2018), 579–591. https://doi.org/10.1016/j.asoc.2017.10.047

[18] Tinglin Huang, Yuxiao Dong, et al. 2021. MixGCF: An Improved Training Method for Graph Neural Network-based Recommender Systems. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery Data Mining (KDD '21)*. Association for Computing Machinery, New York, NY, USA, 665–674.

[19] Bharti Khemani, Ketan Kotecha, and Sudeep Tanwar. 2024. A review of graph neural networks: concepts, architectures, techniques, challenges, datasets, applications, and future directions. *Journal of Big Data* 11 (01 2024). https://doi.org/10.1186/s40537-023-00876-4

[20] Bharti Khemani, Shruti Patil, Ketan Kotecha, and Sudeep Tanwar. 2024. A review of graph neural networks: concepts, architectures, techniques, challenges, datasets, applications, and future directions. *Journal of Big Data* 11, 1 (2024), 18. https://doi.org/10.1186/s40537-023-00876-4

[21] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*.

[22] Ahmed Lamkanfi, Serge Demeyer, Quinten David Soetens, and Tim Verdonck. 2011. Comparing Mining Algorithms for Predicting the Severity of a Reported Bug. In *2011 15th European Conference on Software Maintenance and Reengineering*. 249–258. https://doi.org/10.1109/CSMR.2011.31

[23] Saurabh Malgaonkar, Sherlock A. Licorish, and Bastin Tony Roy Savarimuthu. 2022. Prioritizing user concerns in app reviews – A study of requests for new features, enhancements and bug fixes. *Information and Software Technology* 144 (2022), 106–798. https://doi.org/10.1016/j.infsof.2021.106798

[24] Lionel Marks, Ying Zou, and Ahmed E. Hassan. 2011. Studying the fix-time for bugs in large open source projects. In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering* (Banff, Alberta, Canada) *(Promise '11)*. Association for Computing Machinery, New York, NY, USA, Article 11, 8 pages. https://doi.org/10.1145/2020390.2020401

[25] Tim Menzies and Andrian Marcus. 2008. Automated severity assessment of software defect reports. In *2008 IEEE International Conference on Software Maintenance*. IEEE, 346–355.

[26] G Murphy and Davor Cubranic. 2004. Automatic bug triage using text categorization. In *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*. Citeseer, 1–6.

[27] OpenAI. 2023. ChatGPT (Mar 14 version). https://chat.openai.com/chat. Large language model.

[28] Rama Ranjan Panda and Naresh Kumar Nagwani. 2023. Software bug priority prediction technique based on intuitionistic fuzzy representation and class imbalance learning. *Knowl. Inf. Syst.* 66, 3 (oct 2023), 2135–2164. https://doi.org/10.1007/s10115-023-02000-7

[29] Showmick Guha Paul, Arpa Saha, Md. Zahid Hasan, Sheak Rashed Haider Noori, and Ahmed Moustafa. 2024. A Systematic Review of Graph Neural Network in Healthcare-Based Applications: Recent Advances, Trends, and Future Directions. *IEEE Access* 12 (2024), 15145–15170. https://doi.org/10.1109/ACCESS.2024.3354809

[30] Camelia Serban and Andreea Vescan. 2019. Predicting reliability by severity and priority of defects. In *Proceedings of the 2nd ACM SIGSOFT International Workshop on Software Qualities and Their Dependencies* (Tallinn, Estonia) *(SQUADE 2019)*. Association for Computing Machinery, New York, NY, USA, 27–34. https://doi.org/10.1145/3340495.3342753

[31] Meera Sharma, Punam Bedi, Krishna Kumar Chaturvedi, and V. B. Singh. 2012. Predicting the priority of a reported bug using machine learning techniques and cross project validation. *2012 12th International Conference on Intelligent Systems Design and Applications (ISDA)* (2012), 539–545. https://api.semanticscholar.org/CorpusID:24459126

[32] Mohammed Q Shatnawi and Batool Alazzam. 2022. An Assessment of Eclipse Bugs' Priority and Severity Prediction Using Machine Learning. *International Journal of Communication Networks and Information Security* 14, 1 (2022), 62–69.

[33] Youshuai Tan, Sijie Xu, Zhaowei Wang, Tao Zhang, Zhou Xu, and Xiapu Luo. 2020. Bug severity prediction using question-and-answer pairs from stack overflow. *Journal of Systems and Software* 165 (2020), 110–567.

[34] Yuan Tian, David Lo, and Chengnian Sun. 2013. DRONE: Predicting Priority of Reported Bugs by Multi-factor Analysis. *2013 IEEE International Conference on Software Maintenance* (2013), 200–209. https://api.semanticscholar.org/CorpusID:9704532

[35] Yuan Tian, David Lo, Xin Xia, and Chengnian Sun. 2015. Automated prediction of bug report priority using multi-factor analysis. *Empirical Software Engineering* 20 (2015), 1354–1383.

[36] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. *ArXiv* abs/2302.13971 (2023). https://api.semanticscholar.org/CorpusID:257219404

[37] Qasim Umer, Hui Liu, and Inam Illahi. 2020. CNN-Based Automatic Prioritization of Bug Reports. *IEEE Transactions on Reliability* 69, 4 (2020), 1341–1354. https://doi.org/10.1109/TR.2019.2959624

[38] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in Neural Information Processing Systems* 30 (2017).

[39] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2018. Graph attention networks. *Stat* 1050 (2018), 4.

[40] Zhen Wang, Zhewei Wei, Yaliang Li, Weirui Kuang, and Bolin Ding. 2022. Graph Neural Networks with Node-wise Architecture. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Washington DC, USA) *(KDD '22)*. Association for Computing Machinery, New York, NY, USA, 1949–1958. https://doi.org/10.1145/3534678.3539387

[41] WikiPedia. 2024. Core - MozillaWiki — wiki.mozilla.org. https://wiki.mozilla.org/Core. [Accessed 28-03-2024].

[42] WikiPedia. 2024. UserGuide/BugFields - MozillaWiki — wiki.mozilla.org. https://wiki.mozilla.org/BMO/UserGuide/BugFields#bug_severity. [Accessed 23-03-2024].

[43] Lingfei Wu, Yu Chen, Kai Shen, Xiaojie Guo, Hanning Gao, Shucheng Li, Jian Pei, Bo Long, et al. 2023. Graph neural networks for natural language processing: A survey. *Foundations and Trends® in Machine Learning* 16, 2 (2023), 119–328.

[44] Liang Yao, Chengsheng Mao, and Yuan Luo. 2019. Graph convolutional networks for text classification. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 33. 7370–7377.

[45] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. 2020. Graph neural networks: A review of methods and applications. *AI Open* 1 (2020), 57–81. https://doi.org/10.1016/j.aiopen.2021.01.001

Received 2024-03-28; accepted 2024-04-19