A Hierarchical Attention Networks based Model for Bug Report Prioritization

Anurag Yadav
Department of Computer Science and Engineering,
ABV-Indian Institute of Information Technology and
Management Gwalior, India
bcs_2020013@iiitm.ac.in

Santosh Singh Rathore
Department of Computer Science and Engineering,
ABV-Indian Institute of Information Technology and
Management Gwalior, India
santoshs@iiitm.ac.in

ABSTRACT

Software is becoming increasingly complex, which has led to a surge in bugs. Bug reports typically contain essential details for bug reproduction and resolution. Currently, a triage engineer manually reads bug reports, classifies the bugs, and assigns priorities to them so that critical bugs can be fixed promptly. This task is time-consuming. While machine learning and deep learning models have been developed to automate this task, some have shown subpar performance, and others, although promising, have demonstrated inconsistency when applied to new bug reports. This paper proposes a Hierarchical Attention Network (HAN) model for prioritizing software bug reports. The bug reports are preprocessed through text normalization, tokenization using the DistilBERT tokenizer, and attention mechanisms to capture hierarchical relationships within the text. The model architecture incorporates bidirectional GRU layers and attention mechanisms at both word and sentence levels. The experimental evaluation of the proposed model is done on four software projects using precision, recall, and F1-score measures. Next, the proposed model is compared with convolution neural networks (CNN), support vector machines (SVM), random forests, and logistic regression techniques. The results showed that the proposed HAN model performs better than the other techniques.

KEYWORDS

Software Bug Reports, Bug report Prioritization, DistillBERT, Heirarichal Attention Networks (HAN)

1 INTRODUCTION

Eliminating software bugs is vital, but the sheer volume of bug reports makes resolution time-consuming. For example, the Mozilla project received over 10,000 bug reports in just a few months. Bug tracking systems, such as Bugzilla and Jira, help manage this influx, but the workload often delays critical bug fixes [12]. Bug triagers typically prioritize the bug-fixing process to assist them in solving the problems that are most important to repair in order to prevent this issue. In particular, a bug triager, as an experienced developer, first decides if a new issue is an improvement based on the content of its associated bug report, then assigns it a priority and designates a bug fixer to repair it [17]. The priority in Bugzilla is listed from P1 to P5, with P1 being the greatest priority and P5 denoting the lowest. This process is accurate but labor-intensive. Therefore, the challenge lies in developing an efficient automated bug priority prediction model [2].

Several machine learning and deep learning models have been proposed to overcome the problem of manual bug report prioritization. These models include DRONE [11], cPur [13], and others [9, 14, 15]. While these methods have shown promising results, some have performed poorly, and others, although promising, have been inconsistent when applied to new bug reports. Additionally, most of the models overlook the co-occurrence of global terms and long-distance and nonconsecutive semantics in bug reports, which can lead to inaccurate predictions [5].

To overcome the limitations of existing methods, this work proposes a Hierarchical Attention Networks (HAN) based model for automated bug report priority prediction. The HAN model considers five bug report elements that may help identify priority levels (P1-P5): severity, product, component, description, and summary. First, natural language processing (NLP) techniques are applied to the bug reports to remove unnecessary information. The preprocessed data is then used to construct a heterogeneous graph that contains word nodes and document nodes. This transforms the priority prediction problem into a node classification problem. Since bug reports with different priority levels are often imbalanced, data balancing is applied using oversampling with ADASYN. The proposed model is evaluated on four different datasets, including Mozilla, Eclipse, Netbeans, and GNU Compiler Collection (GCC). Precision, recall, and F1-score metrics are used to assess performance. The proposed model is compared with other state-of-the-art models applied to the bug report prioritization task. The results show that the proposed model outperforms other considered models.

Our contributions are summarized as follows.

- A Hierarchical Attention Networks (HAN) based model is proposed for predicting the priority of bug reports.
- The proposed HAN-based model is combined with data preprocessing techniques, DistilBERT tokenizer, GloVe word embeddings, and ADASYN oversampling to produce better performance.
- The extensive experiments on four large-scale open-source projects, including Mozilla, Eclipse, Netbeans, and GCC are performed.

The rest of the paper is organized as follows. Section 2 describes the background and our motivations for pursuing this work. Section

ISEC'24, Feb 2024, IISC, Bangalore, India
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

3 introduces our approach and uses a bug report example. Section 4 describes how we conducted our experiments and analyzes the results. Section 5 discusses several threats to our approach. Section 6 reviews related work. Section 7 concludes the paper and discusses future work.

2 RELATED WORK

Tan et al. [9] proposed a novel method for determining the severity of bug reports by linking them to the bug repository entry on Stack Overflow. They used three classification algorithms: K-nearest neighbors (KNN), Naive Bayes, and long short-term memory (LSTM). Their experiments showed that their proposed method improved the mean F1-score by 23.03%, 21.56%, and 20.59%, respectively. Xiao et al. [15] proposed DeepLoc, an improved convolutional neural network (CNN)-based model for automatic bug localization. DeepLoc uses CNN-inspired word embedding techniques to represent the functionality of bug reports and source files. They evaluated DeepLoc on 18,500 bug reports from five different projects: AspectJ, Eclipse UI, JDT, SWT, and Tomcat. They compared DeepLoc to four other bug localization techniques: BugLocator, LR+WE, HyLoc, and DeepLoc. Their experiments showed that DeepLoc improved the mean average precision (MAP) for bug localization from 10.87% to 13.1% compared to a regular CNN.

Shakripur et al. [8] proposed a time-based methodology called ABA-TF-IDE, which uses the TF-IDF weighting method. They collected data from the version control system (VCS) software repository, where other project details and source code updates are kept. They trained their model using four machine learning algorithms: support vector machine (SVM), Naive Bayes, vector space model (VSM), and smooth unigram model (SUM). Their results showed that their proposed method performed well, with mean reciprocal ranks (MRR) up to approximately 12% and 9%. Lamkanfi et al. [6] investigated whether it is possible to predict the severity of a ported defect by analyzing its text description. They used a Naive Bayes algorithm on historical data collected from the open-source projects Mozilla, Eclipse, and GNOME. They found that GNOME outperformed Mozilla and Eclipse, with precision and recall ranging from 0.70 to 0.85. Abdelmoez et al. [1] proposed a method to predict the priority of bug reports using a Naive Bayes classifier. They used data from four different systems from the open-source projects Mozilla, Eclipse, and GNOME. The bug reports were ranked in order of mean time to resolution (MTTR). Qasim et al. [14] developed an automated bug report categorization system called DRONE. They used linear regression (LR) to classify priority and obtained an average F1-score of nearly 29%.

Tian et al. [10] proposed a nearest-neighbor-based method for predicting the importance of bug reports. They applied their method to a large dataset of Bugzilla reports containing over 65,000 bug reports. Hui et al. [13] proposed a convolutional neural network-based automatic method for predicting the multiclass priority of bug reports. Their method outperforms state-of-the-art methods and improves the average F1-score by more than 24%. Choudhary et al. [3] developed an SVM-based model for priority prediction that prioritizes Firefox crash reports in the Mozilla Socorro server based on the frequency and entropy of the crashes. Table 2.1 lists a few methods for prioritizing bugs along with their drawbacks.

3 APPROACH FOR BUG REPORT PRIORITIZATION

Figure 1 provides an overview of the proposed bug report prioritization approach. First, we collect a dataset of various bug reports with their priority labels. Next, we preprocess the bug reports by removing comments and other unnecessary information. We then feed the preprocessed text to a tokenizer to generate tokens. The embedding generation layer follows it to generate the embedding vectors of the text. Subsequently, we partition the original dataset into training and testing subsets. We apply a data-balancing technique to balance the representation of all five labels in the training subset. We then supply the training embedding vectors to the HAN model for feature extraction and training. We apply the trained model to the testing subsets to evaluate its performance on bug report priority prediction.

3.1 Input Representation and Preprocessing

The input to the system is the bug reports in the textual format. A bug report contains basic elements, such as description, summary, comment, product, component, priority, severity, assignee, and reporter. Several bug reports are downloaded corresponding to four used open-source projects from Bugzilla and GitHub. The following preprocessing techniques are applied to the acquired datasets.

- Lowercasing: All text was converted to lowercase to ensure uniform word representation.
- Punctuation Removal: Punctuation marks and special characters were removed from the text.
- Stopword Removal: Common stopwords from the English language were removed from the text to focus on meaningful content.
- Stemming: Words were stemmed using the Porter stemming algorithm to reduce them to their root forms.
- Lemmatization: Lemmatization was applied to further normalize words by converting them to their base or dictionary form.

3.2 Tokenization and Embeddings Generation

The input textual report is tokenized using a DistilBERT tokenizer[1], which breaks the text into subwords using WordPiece tokenization. The DistilBERT tokenizer processes segmented text to comprehend contextual information. The output includes token IDs and attention masks, representing subword or word units and highlighting relevant tokens. It works as follows.

DistilBERT uses a subword tokenization approach called WordPiece to divide words into smaller units, allowing the model to handle rare or out-of-vocabulary words effectively. The tokenizer operates using a fixed vocabulary of subword tokens to maintain consistency in tokenization. It assigns a unique ID to each token in the vocabulary and generates attention masks to indicate which tokens are relevant during model training and inference. The DistilBERT tokenizer adds special tokens like CLS and SEP to structure input for various NLP tasks. It encodes text into sequences of token IDs and handles padding to ensure consistent input lengths.

[1]https://keras.io/api/keras_nlp/models/distil_bert/distil_bert_tokenizer/

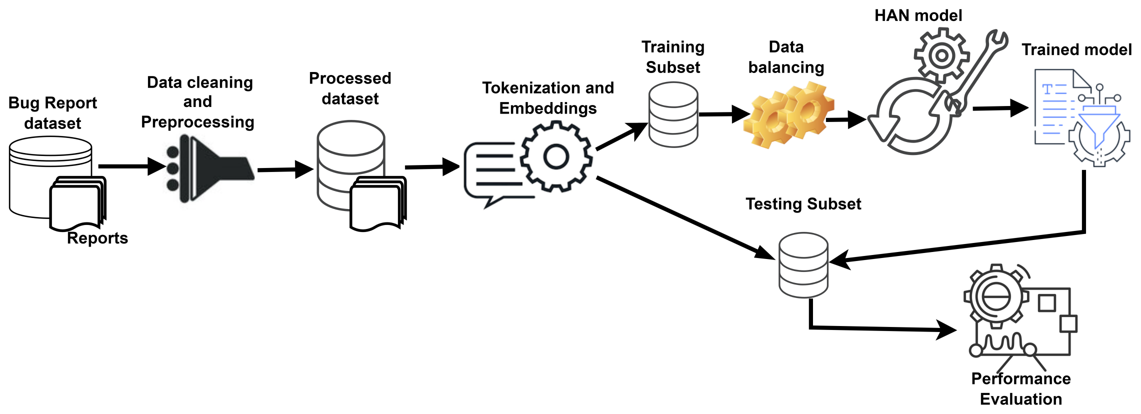


Figure 1: Bug report prioritization approach

The tokenized words are then applied to the GolVe embeddings generation layer. GloVe (Global Vectors for Word Representation) is a word embedding technique that uses word co-occurrence statistics from text data to create word vectors that encode semantic relationships [7]. It works by constructing a matrix capturing word co-occurrence frequencies in a text corpus context window. GloVe then transforms the co-occurrence matrix into word vectors through optimization, aiming to predict co-occurrence probabilities. A loss function measures the difference between predicted and actual co-occurrence probabilities and an iterative process optimizes the word vectors. The learned vectors represent words, preserving semantic relationships and placing similar words closer in vector space.

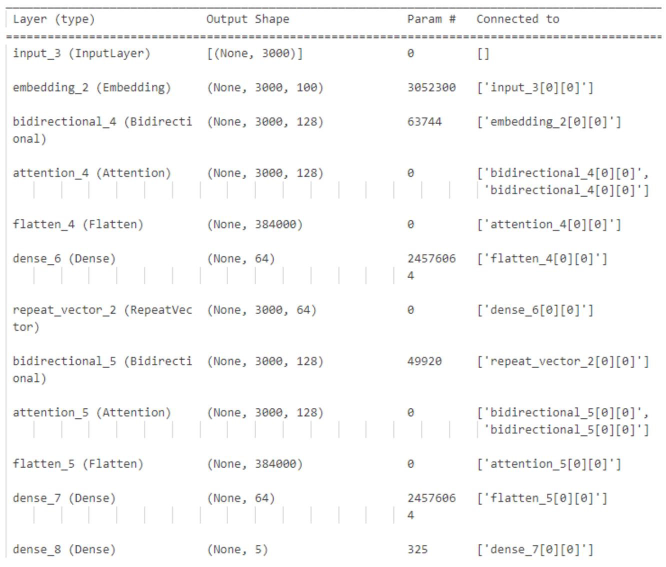
3.3 Data Partitioning and Balancing

The original dataset is partitioned into training and testing subsets in a ratio of 80:20. The training subset has an imbalanced data distribution, with P3 having the highest number of samples. To address this, we used the Adaptive Synthetic (AdaSyn[2]) oversampling technique to balance the training subset. AdaSyn generates synthetic samples of the minority classes, equalizing the data distribution among all labels. However, no data balancing is applied to the testing subset to keep it unbiased for the model assessment.

3.4 Model building using HAN

This work uses Hierarchical Attention Networks (HAN) to build the prediction model [16]. It uses stacked recurrent neural networks on the word level, followed by an attention network. HAN works by first using a Bidirectional RNN to process individual word vectors. An attention network then assigns weights to these vectors based on their importance. The weighted vectors are then summed together to create a sentence vector. This process is repeated at the sentence level to create a document vector. HANs have two levels of attention mechanisms: word-level attention and sentence-level attention. This allows the model to focus on important phrases and clauses while also understanding the context and relationships in the text. The architecture of the used HAN model is depicted in Figure 2.

An input layer equal to the size of *max_sequence_length* is used. It is followed by an embedding layer, which utilizes tokenized words and pre-trained GloVe embeddings provided as weights. Next, a bidirectional RNN (Recurrent Neural Network) layer is used as the word encoder. It processes the embedded words in a bidirectional manner, capturing contextual information from both the past and future words in the sequence. It receives a GRU with 64 units as the argument. After encoding words, an attention mechanism is applied. Attention allows the model to focus on specific words that are more relevant to the context of the entire sequence. In this case, the attention mechanism operates on the word-level RNN output. The output of the attention mechanism is flattened into a one-dimensional tensor. This step simplifies the structure of the data. A dense layer with 64 units follows the flattened output. It performs a linear transformation on the data, adding a degree of non-linearity to the model. Next, the repeat vector layer is used to replicate the word-level information for each word in the sequence. It repeats the word-level representation to match the length of the original sequence.



```
Layer (type)                Output Shape                 Param #   Connected to
==================================================================================================
input_3 (InputLayer)        [(None, 3000)]               0         []

embedding_2 (Embedding)     (None, 3000, 100)            3052300   ['input_3[0][0]']

bidirectional_4 (Bidirecti  (None, 3000, 128)            63744     ['embedding_2[0][0]']
onal)

attention_4 (Attention)     (None, 3000, 128)            0         ['bidirectional_4[0][0]',
                                                                    'bidirectional_4[0][0]']

flatten_4 (Flatten)         (None, 384000)               0         ['attention_4[0][0]']

dense_6 (Dense)             (None, 64)                   2457606   ['flatten_4[0][0]']
                                                         4

repeat_vector_2 (RepeatVec  (None, 3000, 64)             0         ['dense_6[0][0]']
tor)

bidirectional_5 (Bidirecti  (None, 3000, 128)            49920     ['repeat_vector_2[0][0]']
onal)

attention_5 (Attention)     (None, 3000, 128)            0         ['bidirectional_5[0][0]',
                                                                    'bidirectional_5[0][0]']

flatten_5 (Flatten)         (None, 384000)               0         ['attention_5[0][0]']

dense_7 (Dense)             (None, 64)                   2457606   ['flatten_5[0][0]']
                                                         4

dense_8 (Dense)             (None, 5)                    325       ['dense_7[0][0]']
```

Figure 2: Detailed architecture of the developed HAN-based model

[2]https://imbalanced-learn.org/stable/references/generated/imblearn.over_sampling.ADASYN

Similar to the word encoder, all of the above steps are also applied at the sentence level. Another attention mechanism is applied, but it operates on sentence-level representations this time. This allows the model to weigh the importance of different sentences in the text. Similar to the word-level attention output, the sentence-level attention output is flattened. Another dense layer follows the sentence-level flattened output, again applying a linear transformation. The final layer is a dense layer with the same number of units as the number of classes (denoted as *num_classes*). This layer uses a softmax activation function to produce class probabilities. This layer outputs the predicted class probabilities for the input text. The model is compiled using categorical cross-entropy loss, which is common for multi-class classification tasks. The Adam optimizer is used to minimize this loss, and accuracy is chosen as the evaluation metric.

3.5 Model fine-tuning and Prediction

We experimentally fine-tuned the presented models for the given prediction task using the Keras library on Google Colab. In each step, the model was trained on the input data, which consisted of tokenized and encoded sentences. At each step of fine-tuning, the loss was calculated using the categorical cross-entropy (CE) loss function defined in Equation 1. We employed the Adam optimizer for fine-tuning with a learning rate of 0.001 and a decay rate calculated based on the number of epochs. Additionally, early stopping was used to prevent overfitting and ensure the model's ability to generalize well to unseen data.

$$CE = -[y_1.log(\hat{y_1}) + y_2.log(\hat{y_2}) + y_3.log(\hat{y_3}) + ...] \tag{1}$$

Where, y_1, y_2, y_3 are the true labels and $\hat{y_1}, \hat{y_2}, \hat{y_3}$ are predicted labels.

4 EXPERIMENT SETUP

4.1 Dataset

To evaluate the presented HAN-based model, we conducted an experimental analysis on bug reports from four large open-source projects: Mozilla, Eclipse, Netbeans, and GCC. These datasets were prepared by Fang et al. [4] for predicting bug-fixing priority. Table 1 provides a description of the datasets used. All datasets contained five priority levels (P1-P5) and the textual information of the bugs.

Table 1: Description of the Used Bug Report Dataset

Project	Priority	# of BRs	Project	Priority	# of BRs
Mozilla	P1	30,247	Netbeans	P1	11,946
	P2	9,818		P2	18,934
	P3	13,000		P3	18,399
	P4	4,974		P4	2,873
	P5	3,771		P5	0
	Total	**61,810**		**Total**	**52,152**
Eclipse	P1	4,850		P1	3,905
	P2	9,964		P2	12,418
	P3	90,026		P3	16,164
	P4	2,828		P4	384
	P5	3,270		P5	241
	Total	**110,938**		**Total**	**33,113**

4.2 Performance Measures

We employed three performance measures to evaluate the performance of our presented model and compare it with other models.

1. Precision: Precision is the proportion of positive instances (true positives) that are correctly predicted.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

2. Recall: Recall measures the percentage of true positives that were correctly predicted out of all positive cases.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

3. F1-Score: The harmonic mean of recall and precision is known as the F1-score.

$$\text{F1-Score} = \frac{2 \cdot (\text{Precision} \cdot \text{Recall})}{\text{Precision} + \text{Recall}}$$

Here for every priority level i.e. P1, P2, P3, P4, and P5 Precision, Recall and Fl-score are calculated.

4.3 Implementation Details

The HAN model and other ML-based models were implemented with the following details. Text data preprocessing involved using the Natural Language Toolkit[3] (NLTK) for tasks such as converting all text to lowercase, removing mentions, tokenizing the text into individual words, filtering out non-word-tokens-,-and stemming each word to its root form. The DistilBERT tokenizer was created using the Keras library. Similarly, the HAN model was developed using the Keras library, utilizing an ADAM optimizer and a decaying learning rate throughout the epochs. The model was trained for 50 epochs with early stopping criteria. To address the class imbalance issue, the imbalanced-learn Python library was used for class balancing. In the implementation of traditional machine learning models, the scikit-learn library was employed. Bag-of-words (BoW) was used with traditional ML models for feature extraction. These models included Support Vector Machines (SVM), Random Forest, and Logistic Regression. Additionally, a Convolutional Neural Network (CNN) model was developed using the Keras Python library.

5 RESULT AND ANALYSIS

The methodology and experimental procedure outlined in Sections 3 and 4 were employed to construct and evaluate the models. The performance of the models was assessed using three performance measures.

Table 2 reports the results of the presented HAN-based model and other considered models for the bug report priority prediction. From the table, it can be seen that the proposed HAN-based model achieved precision values of 54.84%, 28.57%, 38.71%, 37.14%, and 38.33% for priorities P1, P2, P3, P4, and P5, respectively. Similarly, recall values were 32.69%, 33.33%, 23.08%, 53.06%, and 47.92%, and F1-scores were 40.96%, 30.77%, 28.92%, 43.70%, and 42.59%, respectively. To assess the effectiveness of the proposed model, we conducted a homogeneous comparison with baseline models Logistic Regression, SVM, Random Forest, and CNN. All models used

[3]NLTK Tootkit: https://www.nltk.org/

Table 2: Results for the bug report priority prediction of different models

Priority	Proposed HAN model			CNN Model			SVM			Random Forest			Logistic Regression		
	P (%)	R (%)	F (%)	P (%)	R (%)	F (%)	P (%)	R (%)	F (%)	P (%)	R (%)	F (%)	P (%)	R (%)	F (%)
P1	54.84	32.69	40.96	37.16	4.067	33.83	35.41	36.92	37.02	34.33	36.52	35.52	32.17	33.25	32.70
P2	28.57	43.33	30.77	33.51	21.86	26.46	28.31	21.13	23.54	30.29	20.14	24.89	25.50	20.19	22.54
P3	38.71	23.08	28.92	43.80	44.15	23.54	40.38	40.54	31.22	38.56	43.50	29.53	35.68	38.40	26.99
P4	37.14	53.06	43.70	39.71	39.77	39.74	37.26	38.66	38.39	35.09	39.59	36.76	33.11	27.72	33.85
P5	38.33	47.92	42.59	27.39	30.43	28.83	25.93	27.72	26.52	24.64	27.13	26.09	22.93	25.36	24.08

** P = Precision, R = Recall, F= F1-score (Average values for all projects are reported.)*

the same dataset and preprocessing steps. From the results, we observed that the proposed model outperformed the baseline models in terms of F1 scores for priorities P1, P2, P4, and P5. It also exhibited slightly superior overall performance, as evident in the homogeneous comparison.

6 CONCLUSIONS AND FUTURE WORK

This paper presented a HAN-based model for automated bug report priority prediction. The model used a DistilBERT tokenizer, pre-trained GloVe embeddings, and a HAN architecture at both the word and sentence levels. First, we preprocessed the bug reports and applied data balancing. Then, we fed the processed bug reports into a HAN for model training and prediction. We evaluated the model on four open-source software projects: Mozilla, Eclipse, Netbeans, and GCC. The experimental results showed that the proposed model outperformed other considered machine-learning and deep-learning models.

In the future, we will focus on addressing the issue of imbalanced data. Currently, the representation of priority level P3 is disproportionately higher than the other four levels, introducing bias into the model. To rectify this, we plan to augment the dataset with additional bug reports to achieve a better balance. We will also explore more effective strategies to mitigate this imbalance. Additionally, we are interested in delving into domain-specific bug report prioritization by incorporating bug reports from various domains. This approach will increase the generalizability of the presented model across different domains.

REFERENCES

[1] W. Abdelmoez, M. Kholief, and F. Elsalmy. 2012. Bug fix-time prediction model using naïve bayes classifier. 167–172.

[2] M. Alenezi and S. Banitaan. 2013. Bug reports prioritization: Which features and classifier to use? 2 (2013), 112–116.

[3] P. Choudhary. 2017. Neural network based bug priority prediction model using text classification techniques. *International Journal of Advanced Research in Computer Science* 8 (2017), 15.

[4] Sen Fang, You-shuai Tan, Tao Zhang, Zhou Xu, and Hui Liu. 2021. Effective prediction of bug-fixing priority via weighted graph convolutional networks. *IEEE Transactions on Reliability* 70, 2 (2021), 563–574.

[5] Luiz Alberto Ferreira Gomes, Ricardo da Silva Torres, and Mario Lúcio Côrtes. 2019. Bug report severity level prediction in open source software: A survey and research opportunities. *Information and software technology* 115 (2019), 58–78.

[6] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals. 2010. Predicting the severity of a reported bug. (2010), 1–10.

[7] Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 1532–1543.

[8] R. Shokripour, J. Anvik, Z. M. Kasirun, and S. Zamani. 2015. A time-based approach to automatic bug report assignment. *Journal of Systems and Software* 102 (2015), 109–122.

[9] Y. Tan, S. Xu, Z. Wang, T. Zhang, Z. Xu, and X. Luo. 2020. Bug severity prediction using question-and-answer pairs from stack overflow. *Journal of Systems and Software* 165 (2020), 110567.

[10] Y. Tian, D. Lo, and C. Sun. 2012. Information retrieval based nearest neighbor classification for fine-grained bug severity prediction. 1–10.

[11] Y. Tian, D. Lo, and C. Sun. 2013. Drone Predicting priority of reported bags by multi-factor analysis. 200–209.

[12] Jamal Uddin, Rozaida Ghazali, and H. Shah. [n. d.]. A survey on bug prioritization. *Information and Software Technology* 47 ([n. d.]).

[13] Q. Umer, H. Liu, and I. Illahi. 2020. "CAN"-based automatic prioritization of bug reports. *IEEE Transactions on Reliability* 69, 4 (2020), 1341–1354.

[14] Q. Umer, H. Liu, and Y. Sultan. 2018. Emotion based automated priority prediction for bug reports. *IEEE Access* 6 (2018), 35743–35752.

[15] Y. Xiao, J. Keung, K. E. Bennin, and Q. Mi. 2019. Improving bug localization with word embedding and enhanced convolutional neural networks. *Information and Software Technology* 105 (2019), 17–29.

[16] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alex Smola, and Eduard Hovy. 2016. Hierarchical attention networks for document classification. In *Proceedings of the 2016 conference of the North American chapter of the association for computational linguistics: human language technologies*. 1480–1489.

[17] Jie Zhang, XiaoYin Wang, and H. Mei. [n. d.]. A survey on bug-report-analysis. *Science China Information Sciences* 47 ([n. d.]).