Software Bug Classification Using LLM to Analyze Software Performance Across Multiple Releases

A thesis Submitted in Partial Fulfilment of the Requirements for the Degree of

MASTER OF SCIENCE

in Computer Science

by

Akhlakh Ahmed Reja (Roll: C91/CSC/221003)

Anindita Biswas (Roll: C91/CSC/221004)

Riya Ghosh (Roll: C91/CSC/221018)

Supervised By

Dr. Nabendu Chaki

Professor, Department of Computer Science and Engineering

University of Calcutta

Department of Computer Science and Engineering

Submitted On

June, 2024

ABSTRACT

In the realm of software development, Issue Tracking Systems (ITSs) such as JIRA play a pivotal role in managing and documenting project tasks, often referred to as issues. These systems facilitate an iterative refinement process, allowing each issue to be individually enhanced, discussed, linked, and advanced through the project workflow. Despite the extensive use of JIRA, its comprehensive analysis remains under explored. This study introduces a novel approach to classifying bugs into distinct categories using DeBERTaV3[1],a zero-shot classification Large language models (LLM).

Furthermore, we have used a supervised learning model MultinomialNB to train our dataset with labeled classes and compared the result of the of the supervised model and LLM model.
We performed statistical analysis of the prepared JIRA dataset using the data of the classified bugs to identify bug-fixing trends and patterns across multiple releases of the JIRA software

This research contributes to the existing body of knowledge by providing a comprehensive analysis of issues types in JIRA and introducing a novel approach to bug classification. We believe our findings and the proposed tool will have significant implications for both research and practice in software project management.

Department of Computer Science and Engineering
University Of Calcutta

CERTIFICATE

This is to certify that the dissertation report entitled **Software Bug Classification Using LLM to Analyze Software Performance Across Multiple Releases** was prepared by **Akhlakh Ahmed Reja** (Roll: C91/CSC/221003), **Anindita Biswas**(Roll Number: C91/CSC/221004) and **Riya Ghosh** (Roll Number: C91/CSC/221018) is hereby approved and certified as a creditable study in technological subject carried out and presented in a manner satisfactory to warrant its acceptance for the fulfillment of the Master of Science(M.Sc.) degree in Computer Science from the Department of Computer Science and Engineering, the University of Calcutta, for which it is submitted. It is to be understood that by this approval, the undersigned does not necessarily endorse or approve any statement made, opinion expressed, or, conclusions drawn therein, but approved the project report for the purpose for which it is submitted.

_____

(Dr. Nabendu Chaki)
Project Supervisor

_____ _____

Chairperson PG Board of Studies
Computer Science and Engg.

External Examiner

RECOMMENDATION

It is hereby recommended that the project report entitled **Software Bug Classification Using LLM to Analyze Software Performance Across Multiple Releases** was prepared by **Akhlakh Ahmed Reja** (Roll: C91/CSC/221003), **Anindita Biswas**(Roll Number: C91/CSC/221004) and **Riya Ghosh** (Roll Number: C91/CSC/221018) may be accepted in the fulfillment of the requirements for the degree of 2 years Master of Science, Computer Science from the Department of Computer Science and Engineering, University of Calcutta.

Nabendu Chaki
Project Supervisor
Computer Science and Engineering,
University of Calcutta

ACKNOWLEDGEMENTS

We are thankful to the Department of Computer and Engineering, at the University of Calcutta for allowing us to carry out our project preparation with all the necessary guidance, inspiration, and support. It is a great privilege for us to express our profound and sincere gratitude to our project guide **Nabendu Chaki**, Professor, Department of Computer Science and Engineering, University of Calcutta, for his guidance, valuable assistance, and inspiration throughout the course without which it would have been difficult to complete our project work.

We also express our heartiest gratitude to **Mandira Roy** (senior research fellow) and the faculties of the Department of Computer Science and Engineering and the Department Library for their consistent support in supplying all the necessary resources.

Akhlakh Ahmed Reja	Riya Ghosh	Anindita Ghosh
C91/CSC/221003	C91/CSC/221018	C91/CSC/221004

DECLARATION

We, **Akhlakh Ahmed Reja** (Roll: C91/CSC/221003), **Anindita Biswas**(Roll Number: C91/CSC/221004) and **Riya Ghosh** (Roll Number: C91/CSC/221018), students of M.Sc (Computer Science and Engg.), hereby declare that the Project Dissertation titled —**Software Bug Classification Using LLM to Analyze Software Performance Across Multiple Releases** which is submitted by us to the Department of Computer Science and Engineering, University of Calcutta, Kolkata in partial fulfillment of the requirement for awarding of the Master of Science, is not copied from any source without proper citation. This work has not previously formed the basis for the award of any Degree, Diploma, Fellowship or other similar title or recognition.

_____

Date:

_____

Date:

_____

Date:

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A defect or fault in software is identified as a software bug. A software bug is detected in the software development system at the testing stage of software. A bug indicates the wrong implementation of specified software requirements, or occasionally it may be due to some of the technical restrictions also[7]. The management of bugs in large software projects is typically facilitated through the use of bug tracking tools such as Bugzilla and JIRA. These tools provide various interfaces for logging new bugs, accessing bug information, and updating it, thus enabling effective handling of software bugs in online repositories. Bug management is a critical component of the software development process, encompassing tasks such as prevention, identification, data storage, classification, resolution, prediction, and reporting of bugs. One of the key tasks in bug management is software bug classification, which involves categorizing software bugs into predefined and meaningful categories. Effective classification is crucial for the overall efficiency of bug management.One such possible use of bug classification could be assigning dedicated developers of that bug type categories.

In the recent years, generative models have taken both academia and public attention by storm.The main appeal of text generation is that it is so universal, that almost any other text-related task can be reformulated as a text generation task[9].The advances in Pre-trained

Language Models (PLMs) can created new state-of-the-art results on classification of texts as a natural language processing (NLP) tasks. We have used zero-short-classifiaction method to classify our dataset using a Large Language Model named DeBERTaV3.

Chapter 2

Problem Formulation

Sorting bugs for developers to fix is a challenging and lengthy process. Developers typically specialize in specific areas, such as GUI design or Java functionality. Correctly matching a bug to a developer who is skilled in that area can save time and keep developers engaged by aligning the tasks with their interests. However, it's tough for those who assign bugs to know which developer is best for a particular issue without understanding the bug's category. This study introduces a method to categorize open source software bugs based on the descriptions provided by those who report them.

2.1 Scope and motivation

In this paper, we targeted in Analyzing the service deliverability across multiple releases of a software(here we have used the JIRA repository). We tried doing this in the following steps :
Firstly, identifying issues that are mentioned as failure(Bug) and classifying them into some common categories. For the bug classification problem , we analyze the bug reported in the popular JIRA project, with the aim of building a taxonomy and description of the types of reported bugs; then, we devise and evaluate an automated classification model that is able to classify reported bugs according to the defined taxonomy.Here we have studied with six main common bug types over

the considered systems, including functional bugs, usability/workflow bugs, security bugs, documentation bugs, UI bugs, and performance bugs.
Secondly, we do statistical analysis of the prepared JIRA data set using the data of the classified bugs to identify bug-fixing trends and patterns across multiple releases of the JIRA software.

2.2 Literature survey

Given that the resource requirements for training and deploying generative LLMs are prohibitive for many researchers and practitioners, this paper investigates other types of universal models, that make a different trade-off between resource requirements and universality. The literature has developed several other universal tasks that cannot solve generative tasks (summarization, translation etc.), but can solve any classification task with smaller size and performance competitive with generative LLMs[11].
The principle of universal classifiers is similar to generative models: A model is trained on a universal task, and a form of instruction or prompt enable it to generalize to unseen classification tasks. While several efficient approaches to universal classification exist[10] [11].
Davor Cubranic and Gail C.Murphy have proposed an approach for automatic bug triage using text categorization [5] . They proposed a prototype for bug assignment to developer using supervised Bayesian learning. The evaluation shows that their prototype can correctly predict 30% of the report assignments to developers. The prototype used

theword frequency as input to the classifier. The words were extracted using Natural Language Processing Techniques. The words can be considered as unigram features obtained irrespective of the type of bugs. They analyzed their technique on open source eclipse bug database.
Deqing Wang, Mengxiang Lin, Hui Zhang, Hongping Hu have implemented a tool Rebug-Detector, to detect related bugs using bug information and code features[2]. They extracted features related to bugs and used relationship between different methods that is overloaded or overridden methods. They evaluated Rebug-Detector on an open source project: Apache Lucene-Java. The results show that bug features and code features extracted by their tool are useful to find real bugs in existing projects.

The abstract discusses the study of various link types in JIRA repositories, focusing on their prevalence, characteristics, and implications for link type prediction. The study analyzed 607,208 links connecting 698,790 issues across 15 public JIRA repositories, identifying 75 unique link types. These link types were categorized into five general categories: General Relation, Duplication, Composition, Temporal/Causal, and Workflow. The study observed trends in the structures of these graphs, such as Duplication links typically representing simpler issue graphs and Composition links presenting hierarchical tree structures. The study also evaluated the robustness of two state-of-the-art duplicate detection approaches from the literature on the JIRA dataset, finding that they often confuse between Duplication and other link types. The authors suggest extending the training sets with other

link types to improve the accuracy of these approaches[3].

The paper presents a solution for managing complex software projects using a Jira plug-in, addressing the challenges posed by large numbers of issues and their interdependencies. It starts by noting that issue tracking systems like Jira are commonly used in software development to manage tasks, bugs, and requirements, but maintaining an overview of dependencies can be difficult, especially in long-term projects with many issues[4].

The abstract and introduction of the paper describe the use of Issue Tracking Systems (ITSs) in organizations to document and track project work through issues. ITSs like GitHub, GitLab, and Bugzilla have been extensively studied, but Jira, despite its popularity and rich data, has not received as much research attention due to the scarcity of accessible public datasets. To address this gap, the authors release a comprehensive dataset encompassing 16 public Jira repositories, 1822 projects, and 2.7 million issues, which includes 32 million changes, 9 million comments, and 1 million issue links[6].

The paper discusses the use of software issue trackers, such as Jira, for recording problems and improvement requests in software projects. These issue trackers are vital for managing bug reports, feature requests, and other project-related tasks. However, users often misclassify these issues, such as submitting improvement requests as bugs, which wastes developers' time. The paper investigates the use of ma-

chine learning techniques to automate the classification of issue reports into bugs or non-bugs to alleviate this problem[8].

The paper compares the performance of ten classic machine learning algorithms in classifying software bugs across different bug repositories. The algorithms evaluated include Naïve Bayes, Naïve Bayes Multinomial, Discriminative Multinomial Naïve Bayes (DMNB), J48, Support Vector Machine (SVM), Radial Basis Function (RBF) Neural Network, Classification using Clustering, Classification using Regression, Adaptive Boosting (AdaBoost), and Bagging. These algorithms are applied to four open-source bug repositories: Android, JBoss Seam, Mozilla, and MySQL. The study employs a 10-fold cross-validation technique to evaluate classification accuracy and F-measure for each algorithm. Additionally, it introduces a software bug taxonomy hierarchy with eleven standard bug categories and examines the impact of the number of categories on classifier performance[7].

Chapter 3

Problem Solution

3.1 Dataset

Issues of type of Jira repository[13] has been collected form Atlasian Jira - an open source repository. Data are stored in JSON (Javascript object notation) format , cleaned, and processed for the Large Language Model. The retrived issue descriptions are srored in a csv file with labled classes for the *naïve bayes multinomial (NBM)* .

Reason for using JIRA Repository

- **Popularity and Widespread Use:**
  JIRA is one of the most popular issue-tracking and project management tools used in both open-source and commercial software development projects. Its widespread adoption means that the data collected is highly relevant and representative of real-world software development practices.

- **Public Availability and Accessibility:**
  Open Data: Many open-source projects using JIRA make their data publicly available, providing a valuable resource for academic and industrial research.
  Standardization: The standardized format of JIRA data allows

for easy comparison and aggregation across different projects and studies.

- **Enhancing Software Engineering Research**
  Bug Classification: JIRA datasets allow researchers to classify bugs into different categories, improving understanding of bug types and their impact on software development.
  Issue Management: By studying issue links and relationships, researchers can develop better tools and techniques for issue management, prioritization, and resolution

- **Rich and Detailed Data**
  Comprehensive Records: JIRA records a wide range of information for each issue, including titles, descriptions, comments, status changes, priorities, assignees, and custom fields. This rich data set provides a wealth of information for detailed analysis.
  Historical Data: JIRA maintains historical records of all changes made to issues, which is valuable for studying the evolution of software projects over time.

3.2 Model for prediction

3.2.1 Supervised Model

The task of text (Bug description) classification can be approached from a Bayesian learning perspective, which assumes that the word distributions in documents are produced by a specific parametric model, and the parameters can be estimated from the training data. Multinomial Naive Bayes implements the Naïve Bayes algorithm for multinomial

data. Steps in building a Multinomial model are as follows: first, defining the vocabulary, the number of words which provides the dimension of the feature vector, second, scan the training set to attain following counts- number of documents, number of documents of class for all classes, the frequency of word in document for all words and third, approximate likelihoods and priors once the training is performed and parameters are ready. For every new unlabelled document, the posterior probability for every class is estimated[7].

3.2.2 Large Language Model

Zero-shot-classification

Zero-shot text classification poses a challenge in predicting class labels for text instances without requiring labeled instances for supervised training. Effective solutions to this problem are crucial for many real-world applications, as it diminishes the labor-intensive process of manual labeling. With the remarkable advancements of large language models (LLMs) in recent years, exploiting the generative capabilities of such models to tackle zero-shot text classification problems has emerged as a critical research question. Recent research in zero-shot text classification primarily falls into two distinct groups. The first approach applies LLM (with billions of parameters) in label prediction with the help of human instructions or prompts. The second approach to zero-shot classification involves the self-training of smaller language models, often comparable in size to BERT. In these methods, the models predict "pseudo labels" for unlabeled instances, and then use these instances alongside their assigned pseudo labels as supervised data for model fine-

tuning. This process is iterated for the model to incrementally adapt to the target domain [12].

DeBERTaV3

DeBERTa improves BERT with two novel components: DA (Disentangled Attention) and an enhanced mask decoder. Unlike existing approaches that use a single vector to represent both the content and the position of each input word, the DA mechanism uses two separate vectors: one for the content and the other for the position. Meanwhile, the DA mechanism's attention weights among words are computed via disentangled matrices on both their contents and relative positions. Like BERT, DeBERTa is pre-trained using masked language modelling. The DA mechanism already considers the contents and relative positions of the context words, but not the absolute positions of these words, which in many cases are crucial for the prediction. DeBERTa uses an enhanced mask decoder to improve MLM by adding absolute position information of the context words at the MLM decoding layer. Now, DeBERTaV3, which improves DeBERTa by using the RTD training loss of Clark et al. (2020) and a new weight-sharing method. Since RTD in ELECTRA and the disentangled attention mechanism in DeBERTa have proven to be sample-efficient for pre-training, a new version of DeBERTa, referred to as DeBERTaV3, by replacing the MLM objective used in DeBERTa with the RTD objective to combine the strengths of the latter. The performance of DeBERTaV3 can be further improved by replacing token Embedding Sharing (ES) used for RTD, originally proposed in Clark et al. (2020), by a new Gradient-Disentangled Em-

bedding Sharing (GDES) method [1].

3.3 Classification criteria

The goal is to study the effect of number of classes to which the dataset is classified. The number of classes (categories) are one of the decisive parameter in the classification. If small number of classes is taken in classification then accuracy values may be higher but classification result may not be fruitful, vis-à-vis higher number of classes may be practically useful but will be giving very less accuracy. For example if number of classes in classification is one, the accuracy will be 100%, but this classification will not be practically fruitful. The goal of studying the effect of number of classes is to observe the effect of performance parameter values for the different classes and provide a reference to the readers for selecting suitable number of classes (optimum number of classes) for the classification task. It is experimentally shown in the presented work that with increase in the number of classes the accuracy and F-measures for the classification decreases [7].

3.3.1 Categories

Here in our paper, we have classified the observed bugs into six main categories. The below table shows the categorized bugs and their descriptions:

Bug Types	Bug Description
Functional	Functional bugs are associated with the functionality of a specific software component. For example, a Login button doesn't allow users to login, an Add to cart button that doesn't update the cart, a search box not respond to a user's query, etc.In simple terms, any component in an app or website that doesn't function as intended is a functional bug.
Usability/Workflow Bug	Usability defects are present when a software application is inconvenient to operate or negatively impacts the user experience.Workflow bugs are associated with the user journey (navigation) of a software application.
Security Bug	These are persistent and concerning issues in the world of technology. These bugs refer to vulnerabilities or flaws in software, hardware, or systems that malicious actors can exploit to gain unauthorized access, steal sensitive information, disrupt services, or cause other harmful consequences
UI Bug	A UI bug is a problem or error in the user interface of a software application that prevents it from functioning correctly or efficiently. These bugs can be related to visual elements, navigation, interaction, or other aspects of the user interface
Performance Bug	It is mainly related to the stability, speed, or response time of software resources. Any defect that undermines these features falls into the performance bug category.This type of programming defect causes significant performance degradation and leads to a frustrating user experience.Performance bugs usually don't generate incorrect results or crashes in the program under test. Therefore, they cannot be detected simply by checking the software output.
Documentation Bug	Documentation bugs refer to errors, inconsistencies, or omissions in the documentation associated with software development and usage. These bugs can occur in various types of documentation, such as user manuals, API documentation, installation guides, help files, release notes, and other related materials.

Table 3.1: Bug Descriptions

Chapter 4

Experiments

4.1 Naïve bayes multinomial (NBM)

The Multinomial Naive Bayes (MNB) model is a probabilistic learning method commonly used in Natural Language Processing (NLP). The model is based on the Bayes theorem and assumes that the features (in this case, the words in the bug descriptions) are conditionally independent given the class label (bug type).
Given a document (d) (a bug description in our dataset), the goal is to find the class (c) (bug type) that maximizes the posterior probability (P(c | d)).
According to Bayes' theorem:

$$P(c \mid d) = \frac{P(d \mid c) \times P(c)}{P(d)}$$

Where:

- (P(c | d)) is the posterior probability of class (c) given document (d).
- (P(d | c)) is the likelihood of document (d) given class (c).
- (P(c)) is the prior probability of class (c).
- (P(d)) is the evidence, a scaling factor that ensures probabilities sum to 1.

In practice, the models compute the log probabilities to avoid underflow:

$$\log P(c \mid d) = \log P(d \mid c) + \log P(c) - \log P(d)$$

For Multinomial Naive Bayes, (P(d | c)) is computed as:

$$P(d \mid c) = \prod_{i=1}^{n} P(t_i \mid c)^{x_i}$$

Where (t_i) is a term (word) in document (d), (x_i) is the frequency of (t_i) in (d), and (P(t_i | c)) is the conditional probability of term (t_i) given class (c).

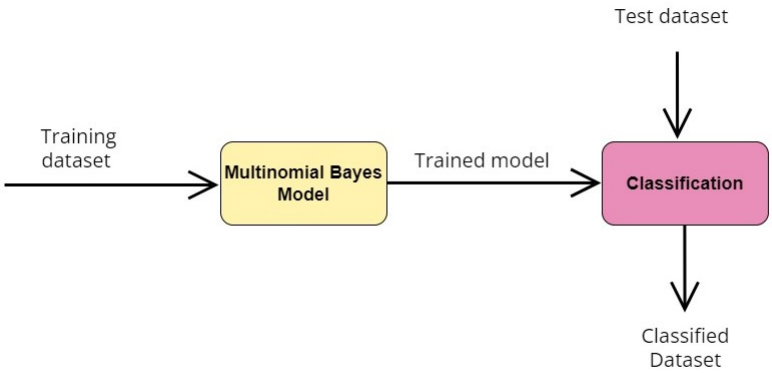

Figure 4.1: Naive Bayes Multinomial

4.1.1 Input

A csv file having two columns a) Bug Descriptions and b) Bug Type.

4.2 DeBERTa-v3

The DeBERTa V3 base model comes with 12 layers and a hidden size of 768. It has only 86M backbone parameters with a vocabulary containing 128K tokens which introduces 98M parameters in the Embedding layer. This model was trained using the 160GB data as DeBERTa V2. DeBERTa-V3 classifier is trained on 33 datasets with 389 diverse classes.

Fine-tuning on NLU tasks

Model	**Vocabulary(K)**	**Backbone Params(M)**	**SQuAD 2.0(F1/EM)**	**MNLI-m/mm(ACC)**
DeBERTa-v3-base	128	86	**88.4/85.4**	**90.6/90.7**

Table 4.1: Performance measurment of DeBERTa-v3

Above table shows the dev results on SQuAD 2.0 and MNLI tasks [1]. The attention mechanism in DeBERTa can be represented as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + S\right)V$$

Where (Q, K, V) are the query, key, and value matrices derived from the input embeddings, (d_k) is the dimensionality of the keys, and (S) is the relative position matrix.

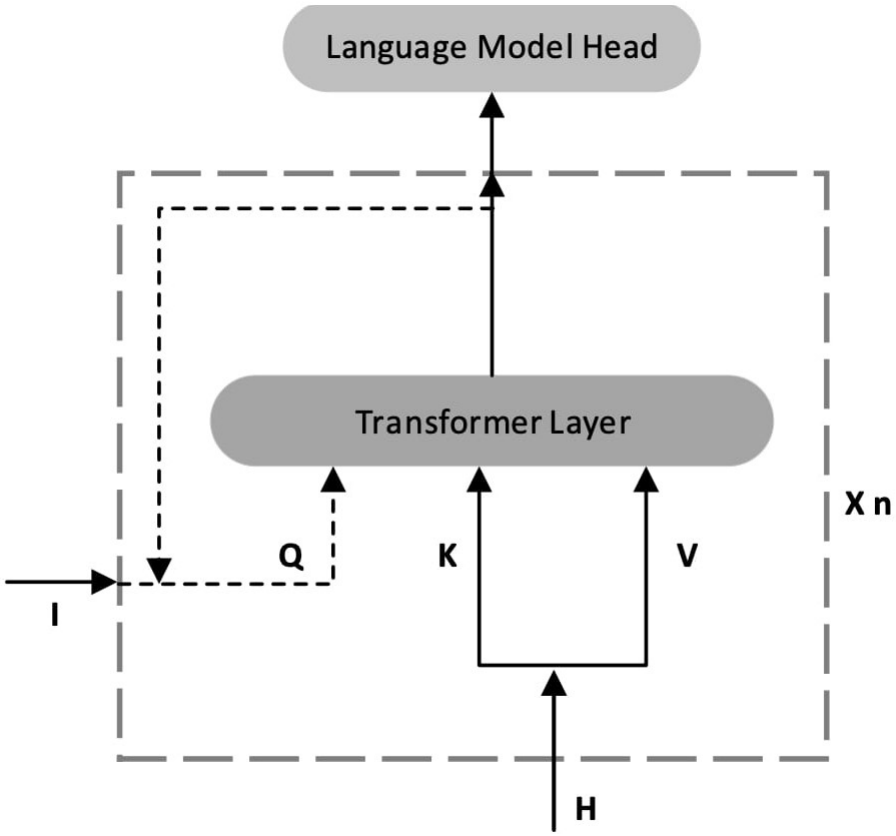

Figure 4.2: Enhanced Mask Decoder

4.2.1 Input

The model accepts a javascript notation file that contains array of list named “issue” , each array list contains key-value pairs as shown in fig. 4.3 .

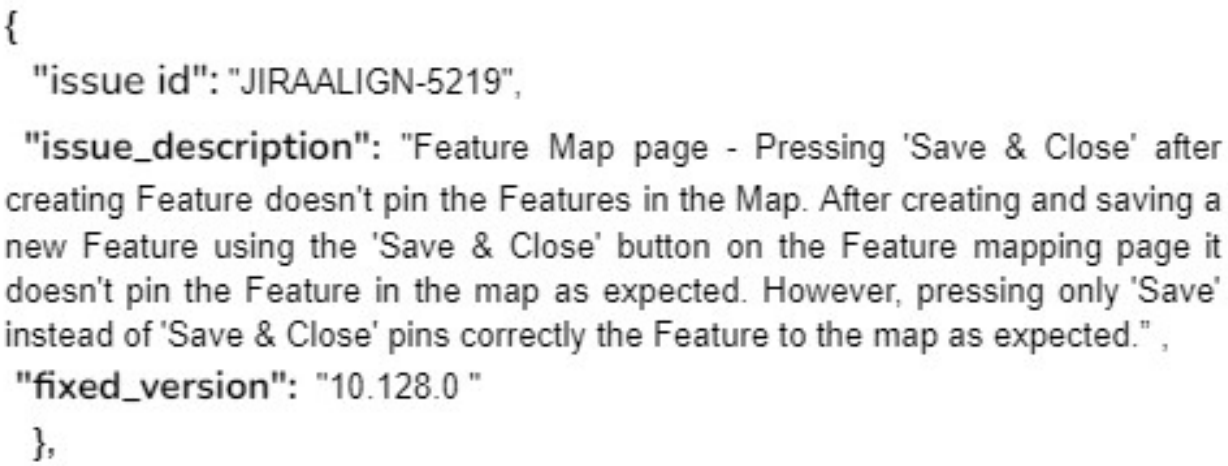

Figure 4.3: example input fro DeBERTa

Chapter 5

Result and Analysis

5.1 Results of NBM

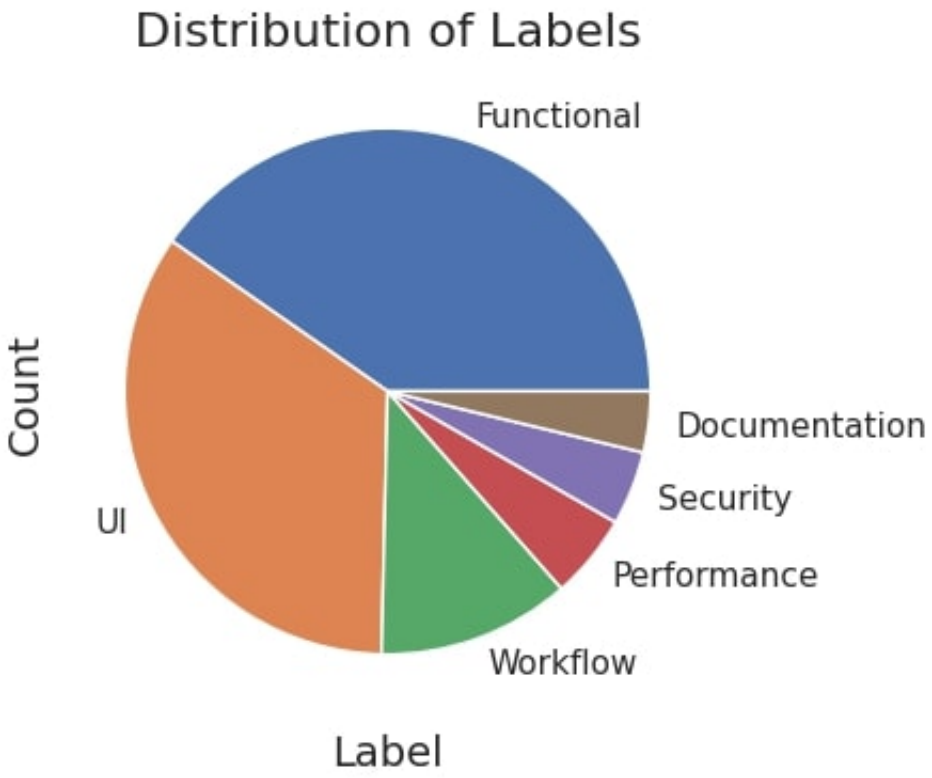

Figure 5.1: Distribution of Bug Types

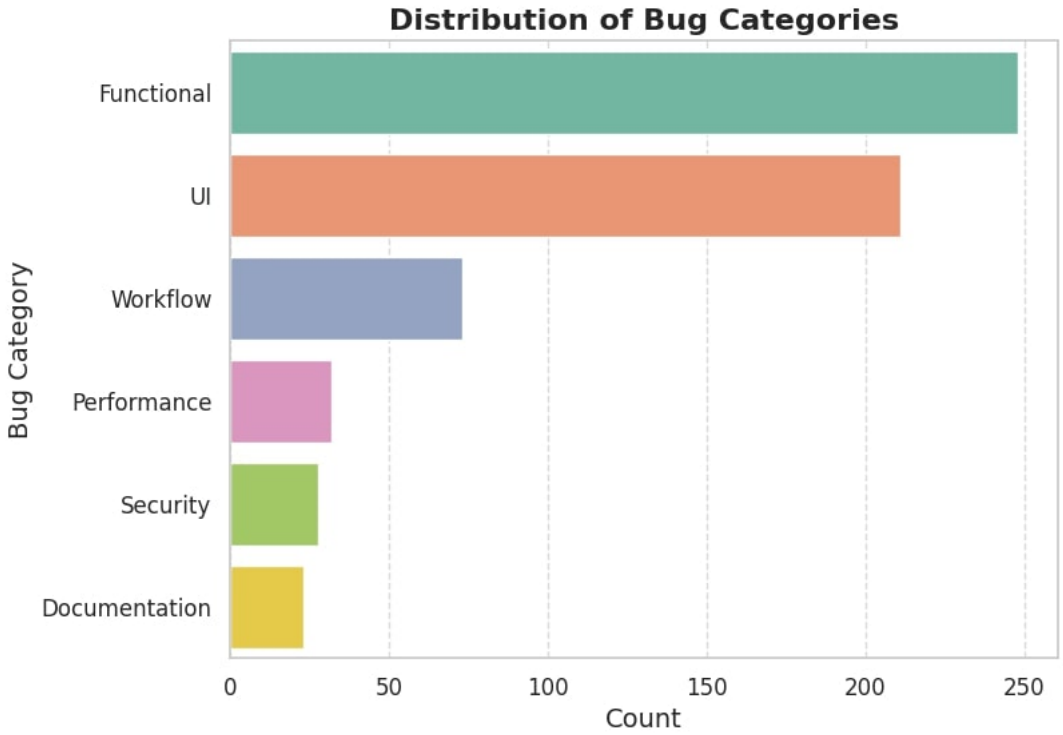

Figure 5.2: Count of Bugs

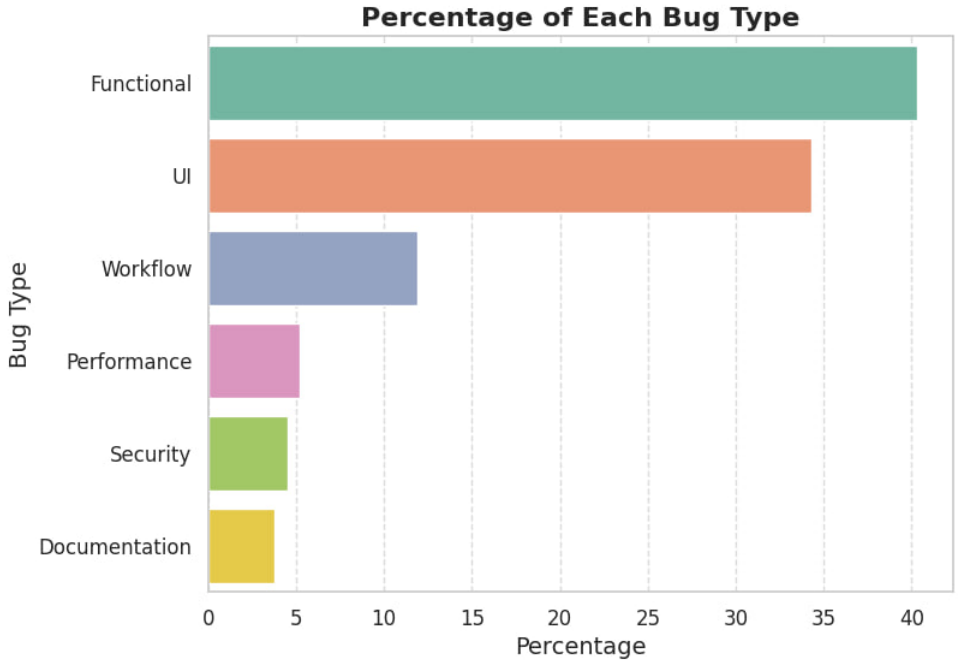

Figure 5.3: Percentage of Bugs

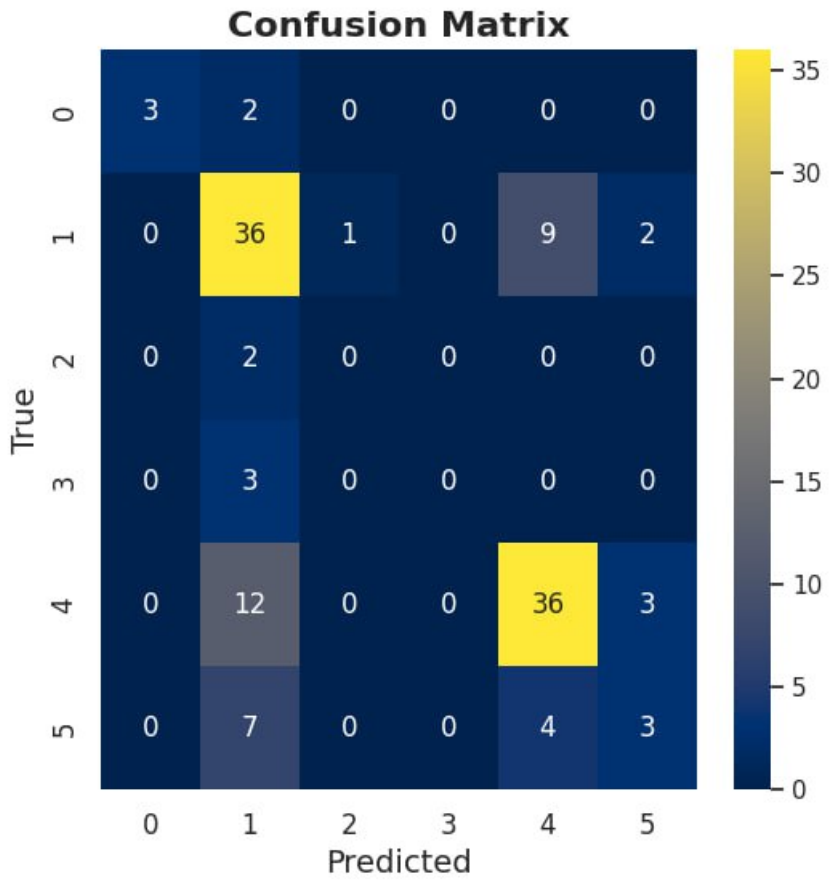

Figure 5.4: confusion matrix

The accuracy A of a classification model can be calculated using the formula:

$$A = \frac{TP + TN}{TP + TN + FP + FN}$$

where:

- TP (True Positives) is the number of positive instances correctly classified.
- TN (True Negatives) is the number of negative instances correctly classified.

- FP (False Positives) is the number of negative instances incorrectly classified as positive.
- FN (False Negatives) is the number of positive instances incorrectly classified as negative.

We have achieved **68%** accuracy.

5.2 Results of DeBERTa-V3

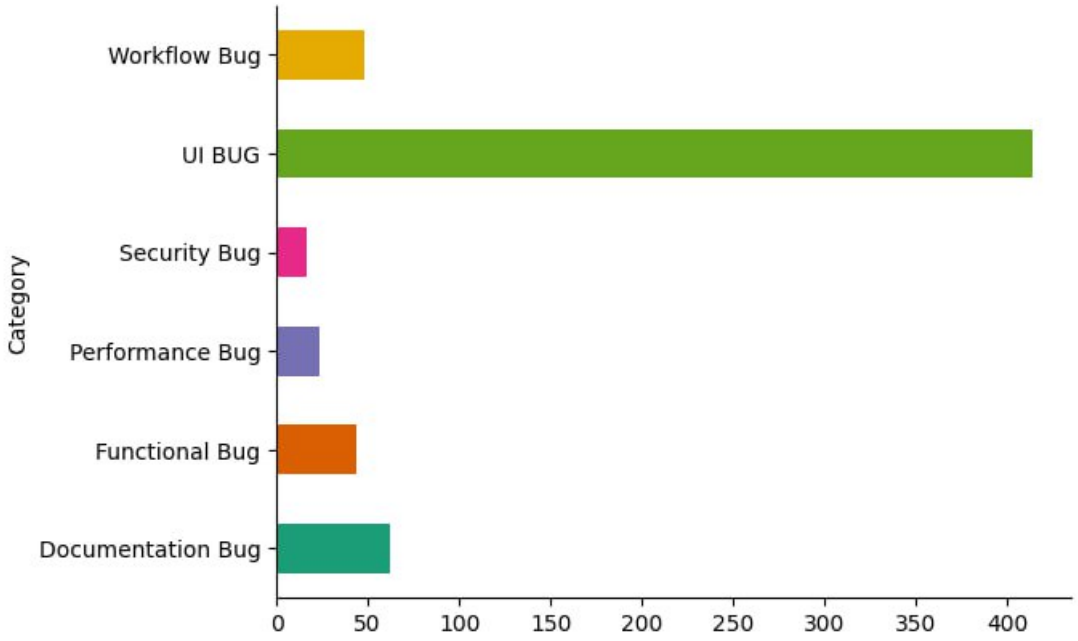

Figure 5.5: Category and count of bugs

Computation of Accuracy: Accuracy is calculated as the number of correct predictions divided by the total number of predictions:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}$$

Output:

The DeBERTa-V3 Model classified each description based on the prediction value. For example -

UI BUG: 0.2496

Logical Bug: 0.1719

Workflow Bug: 0.1575

Functional Bug: 0.1544

Documentation Bug: 0.1484

Security Bug: 0.1183

Category - UI BUG

Prediction Score Calculation:

- For zero-shot classification, DeBERTa uses the entailment feature from models trained on datasets like MNLI. It constructs a premise from the input text and hypotheses from the combination of input text and each candidate label.
- The model outputs probabilities of entailment for each label, which are used as the prediction scores.

In the above example the UI BUG have the highest prediction score thus the model classified the description in UI BUG.

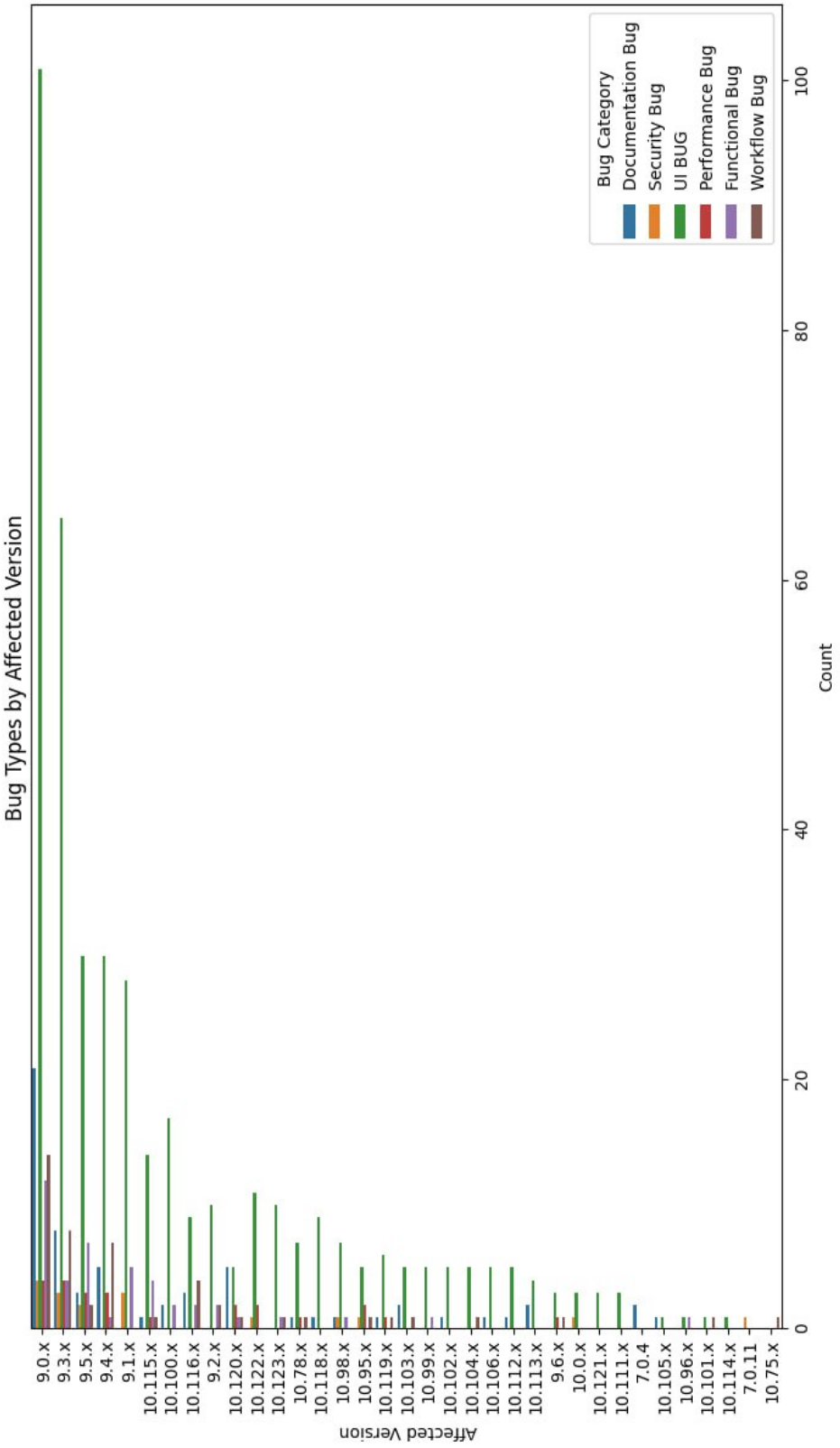

Figure 5.6: Bug Types in Affected Version

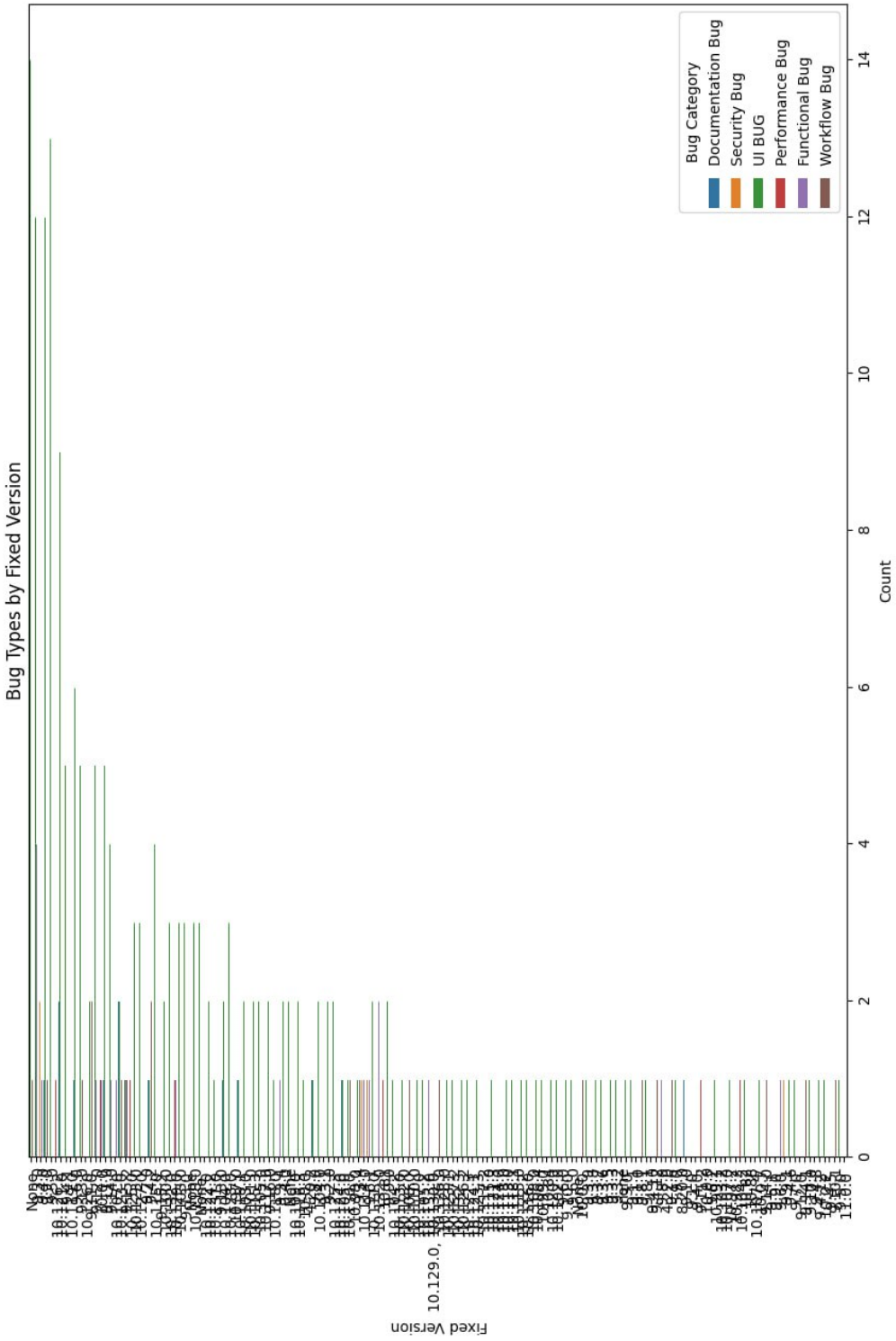

Figure 5.7: Bug Types in Fixed Version

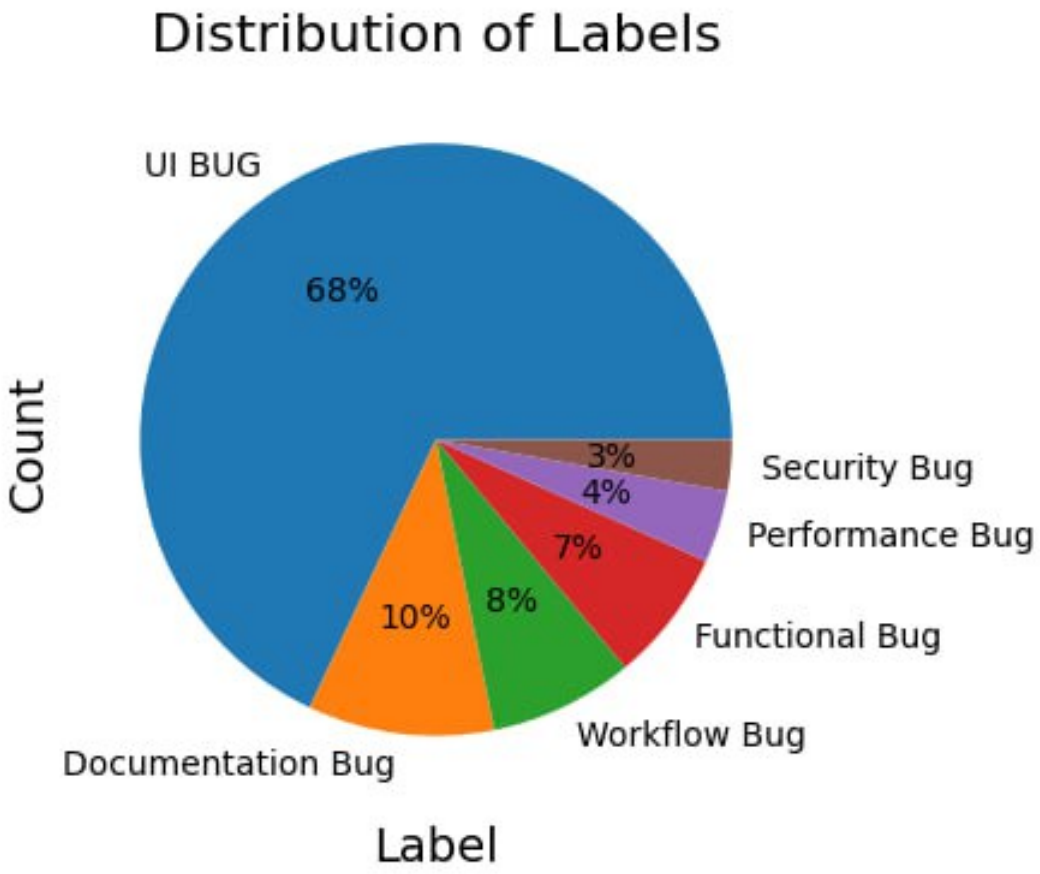

Figure 5.8: Category and count of bugs

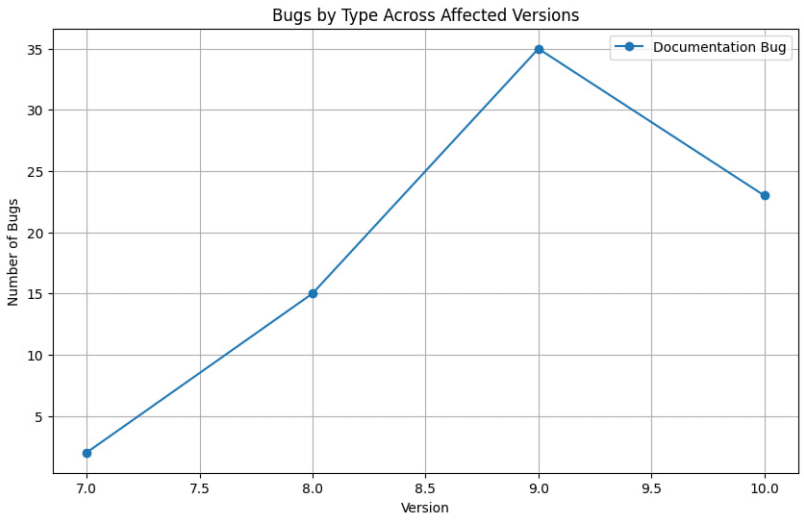

Figure 5.9: Trend of Document bugs over Affected versions

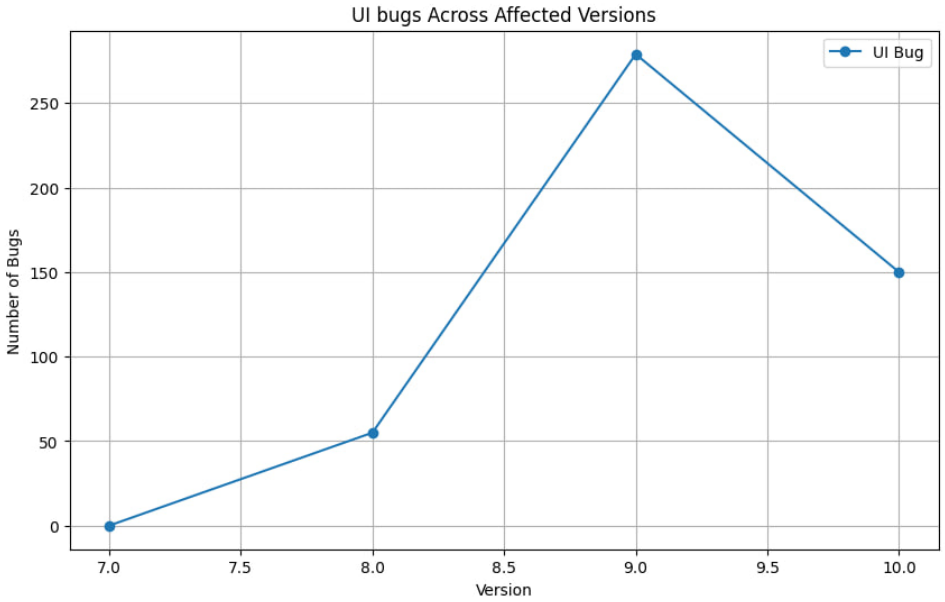

Figure 5.10: Trend of UI bug over Affected versions

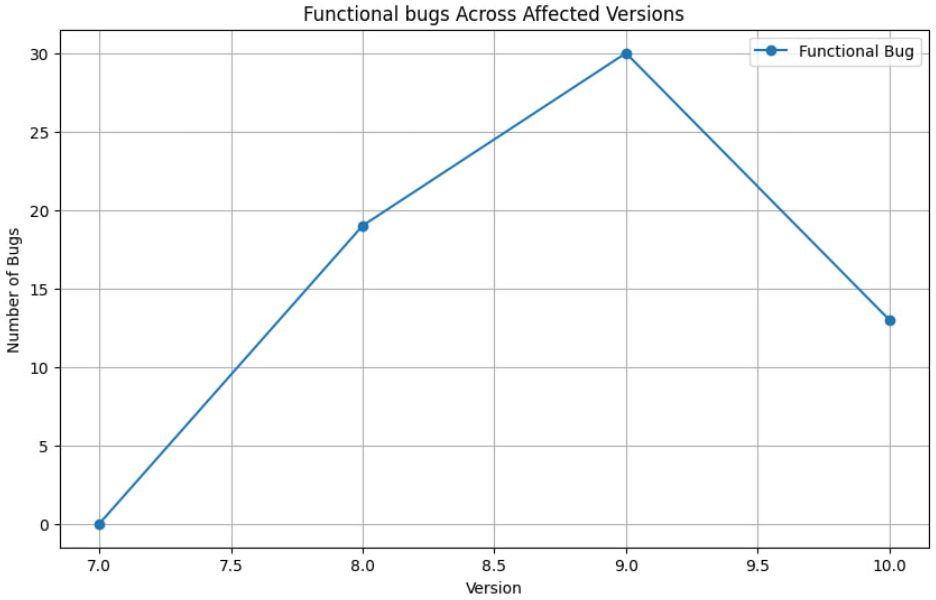

Figure 5.11: Trend of Functional bug over Affected versions

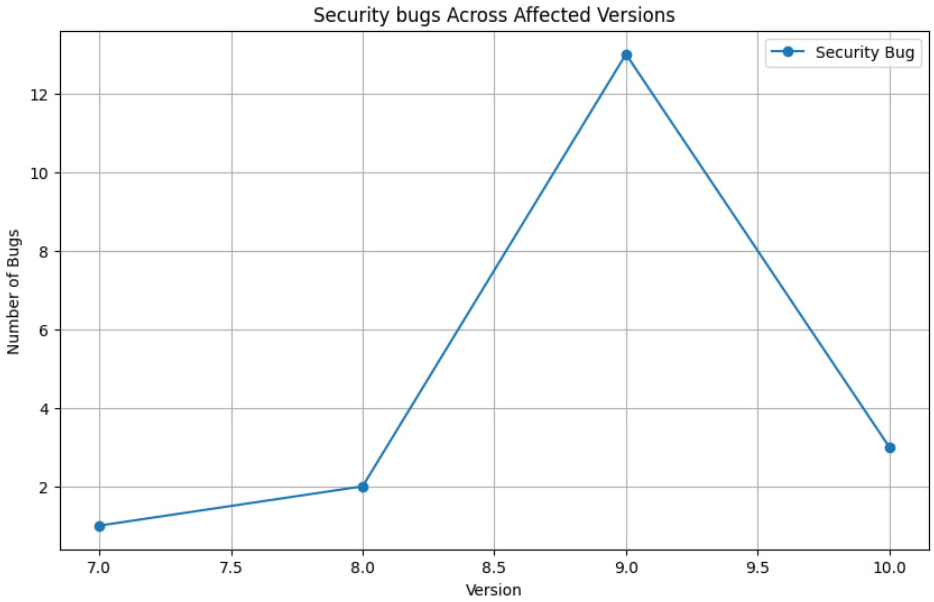

Figure 5.12: Trend of Security bug over Affected versions

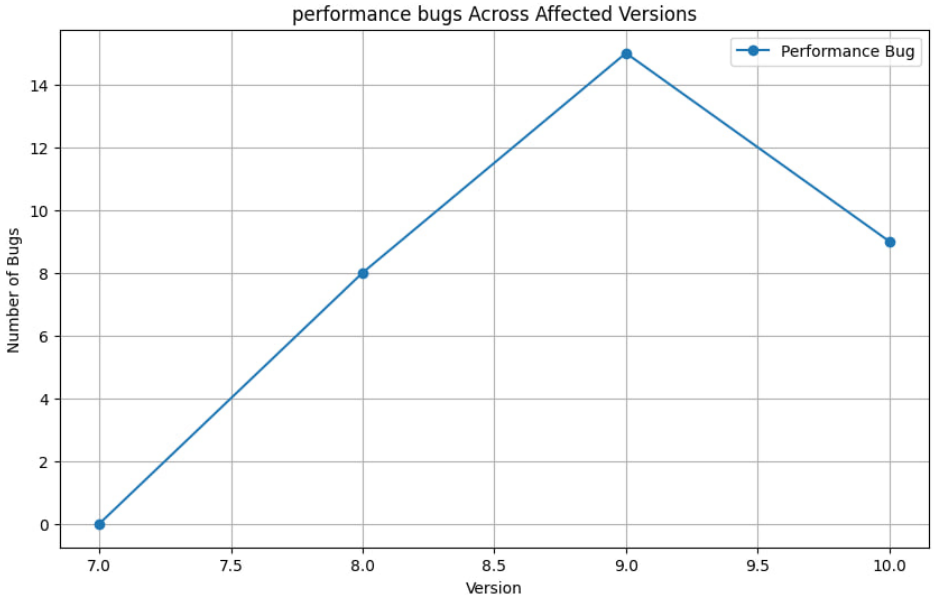

Figure 5.13: Trend of Performance bug over Affected versions

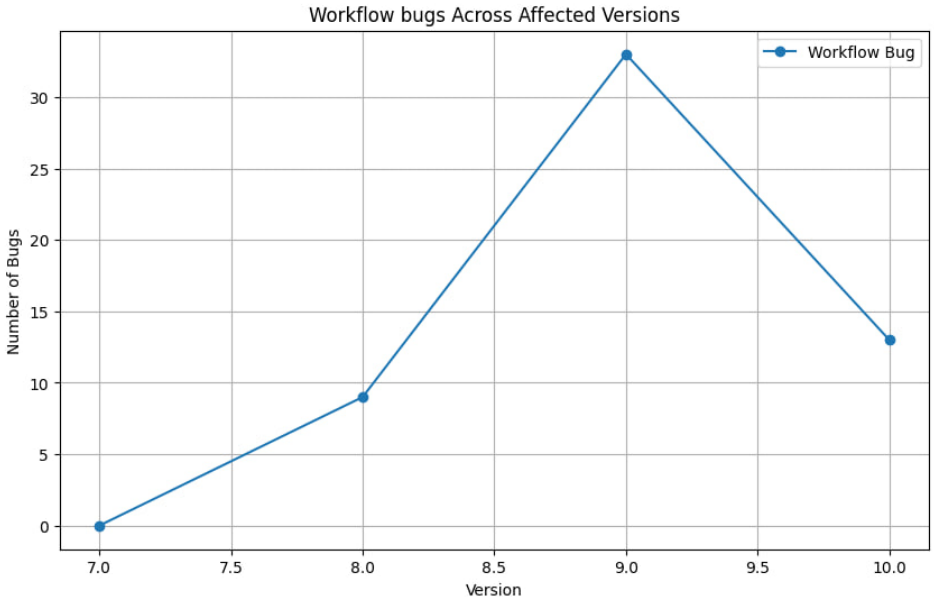

Figure 5.14: Trend of Workflow bug over Affected versions

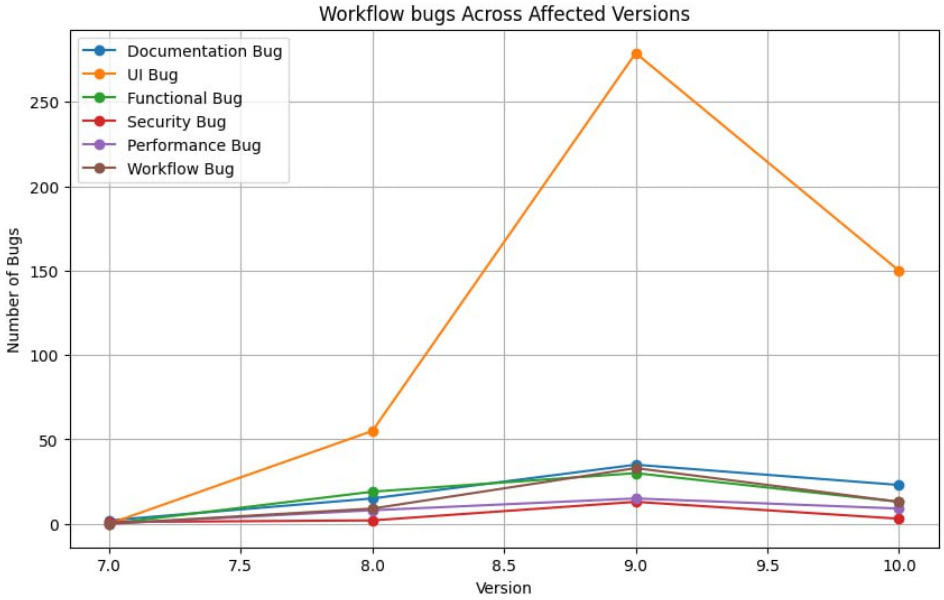

Figure 5.15: Trend of bugs over Affected versions

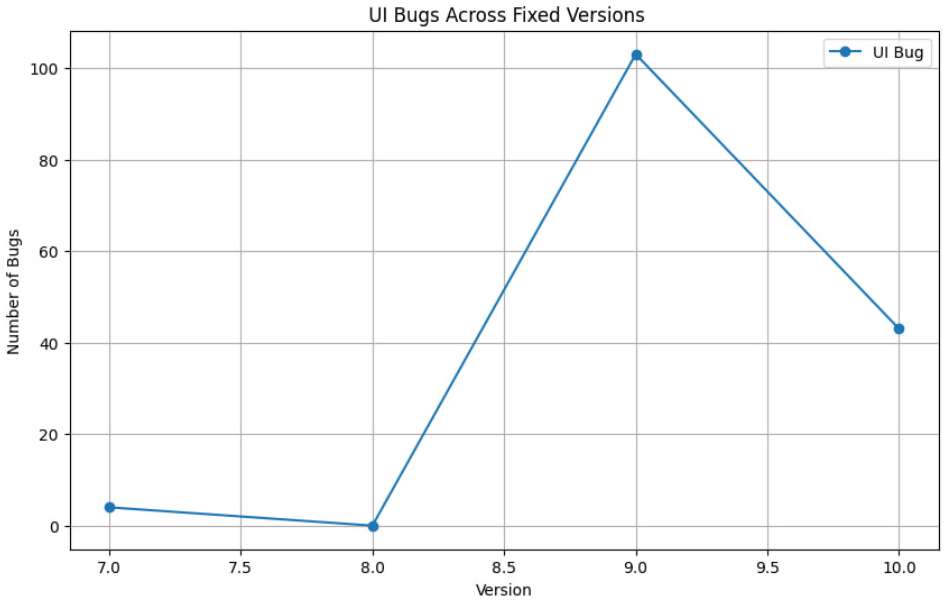

Figure 5.16: Trend of UI bugs over Fixed versions

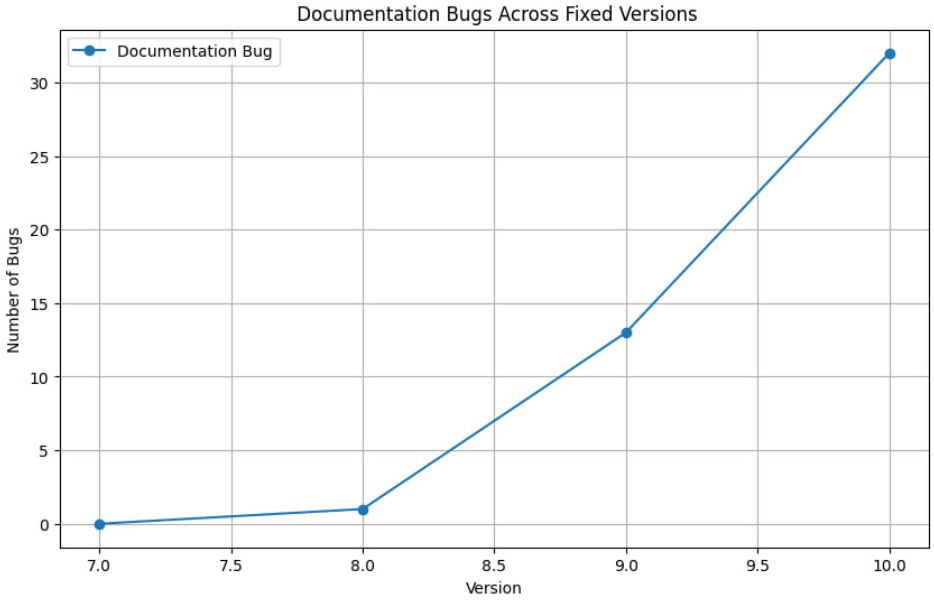

Figure 5.17: Trend of Documentation bugs over Fixed versions

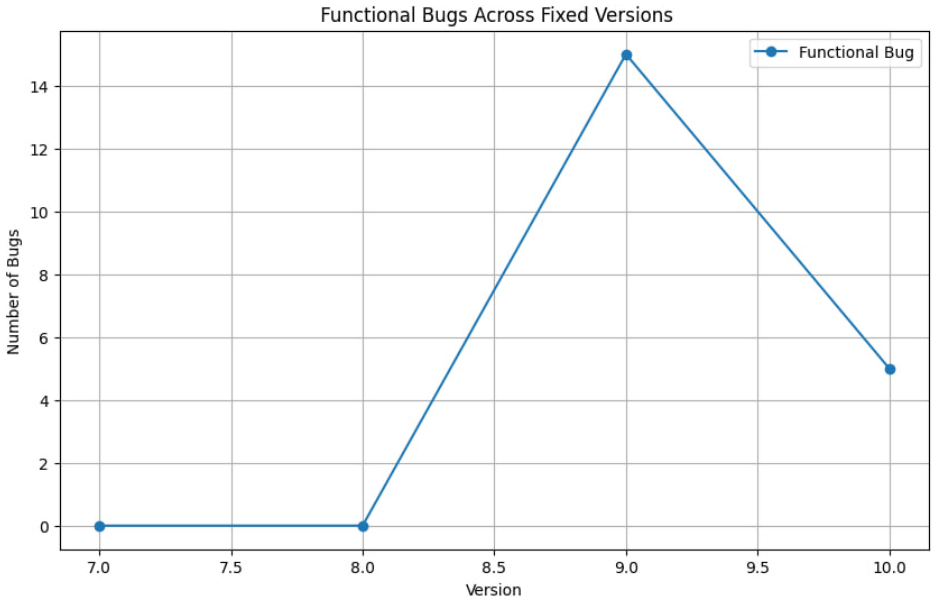

Figure 5.18: Trend of Functional bugs over Fixed versions

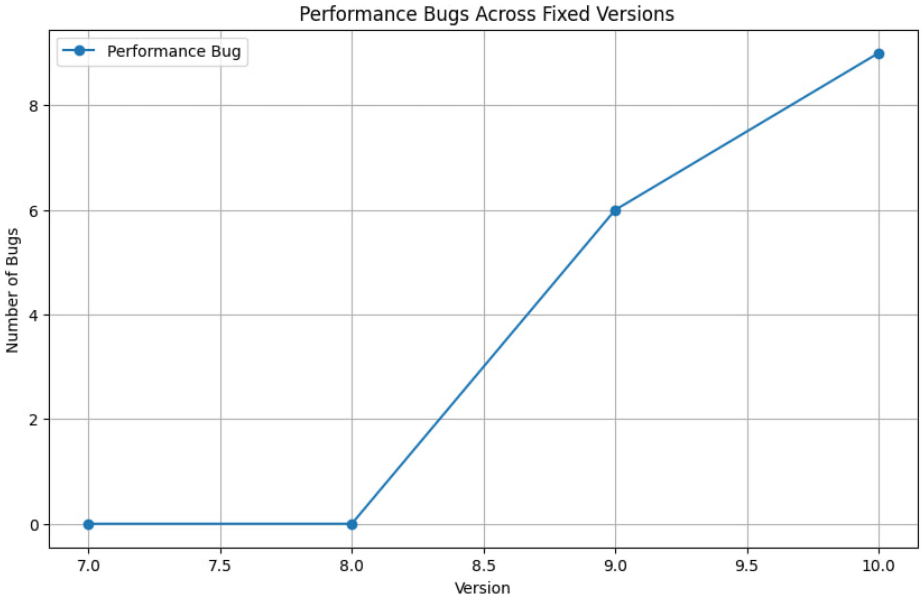

Figure 5.19: Trend of Performance bugs over Fixed versions

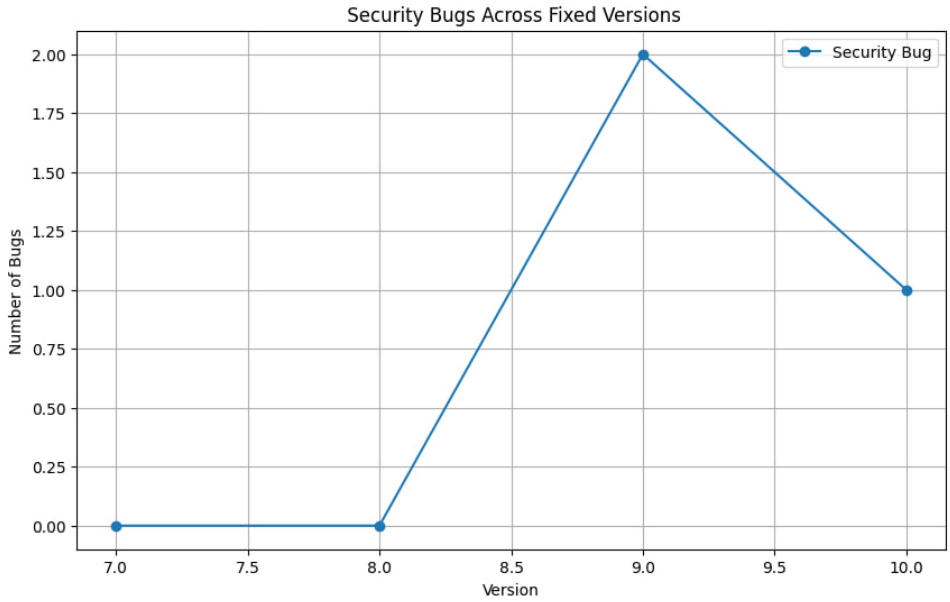

Figure 5.20: Trend of Security bugs over Fixed versions

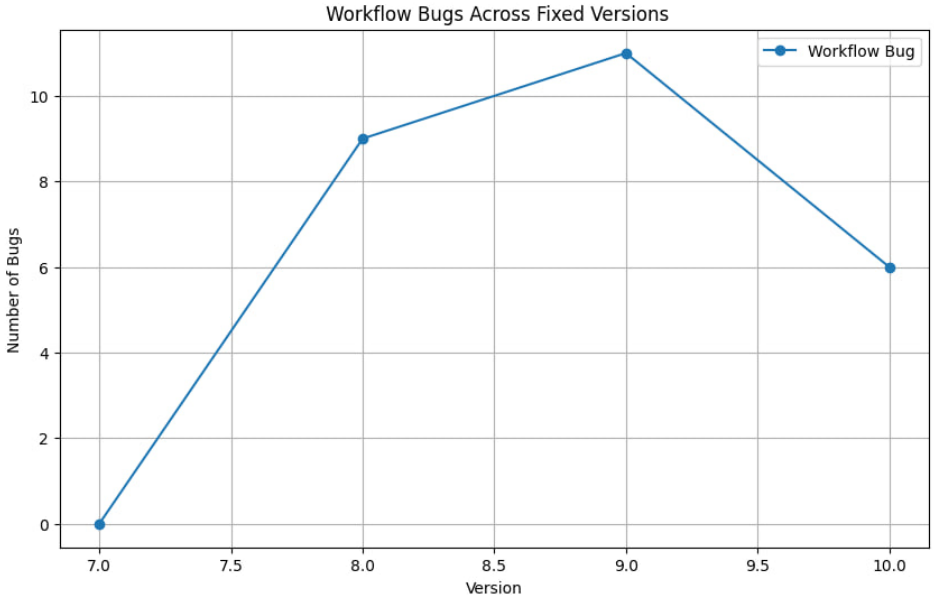

Figure 5.21: Trend of Workflow bugs over Fixed versions

Chapter 6

Conclusions

The Challenge of Bug Triage in Open Source Projects

Open source software thrives on the contributions of its user community. However, this also presents a challenge: efficiently managing bug reports submitted by users with varying technical expertise. Triaging these bugs, which involves classifying them and assigning them to the appropriate developers, is a time-consuming and often tedious task.

Automated Bug Classification

A Solution This research proposes a novel approach to streamline bug triage in open-source projects. By leveraging the power of automated bug classification, the system aims to alleviate the burden on human maintainers.

The Power of LLM Models

The proposed system employs a LLM text classifier and comapres its outcome with a supervised model to find the accuracy. By analyzing the textual content of bug reports, the classifier can learn to categorize them into predefined classes representing different bug types (e.g., user interface issues, security vulnerabilities, etc.).

Promising Results: Towards Efficient Bug Management the system demonstrate promising results, with a reported prediction accuracy of up to 73%. This suggests that automated bug classification can significantly improve the efficiency of bug triage in open-source projects.

By automatically assigning classes to incoming bug reports, the system can expedite the process of assigning bugs to relevant developers for faster resolution.

Chapter 7

Future Work

Looking Ahead: Further Refinement and Integration

While the current research showcases a successful application of Large Language model for bug classification, there's always room for improvement. Future work could involve:

Expanding the Feature Set: Exploring additional features beyond text data, such as code snippets or system logs, could further enhance classification accuracy.

Incorporating Active Learning: Implementing active learning techniques can allow the system to identify the most informative bug reports for human labeling, improving its learning efficiency over time.

Integration with Bug Tracking Systems: Seamless integration with existing bug tracking systems can streamline the bug triage workflow for developers.By continuing to refine and develop this automated bug classification system, we can significantly contribute to more efficient and effective bug management within the open-source software community.

References

[1] Pengcheng He, Jianfeng Gao, and Weizhu Chen. Debertav3: Improving deberta using electra-style pre-training with gradient-disentangled embedding sharing. *arXiv preprint arXiv:2111.09543*, 2021.

[2] CJ Kaufman. Rocky mountain research lab. *Boulder, CO, private communication*, 34:35, 1995.

[3] Clara Marie Lüders, Abir Bouraffa, and Walid Maalej. Beyond duplicates: Towards understanding and predicting link types in issue tracking systems. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 48–60, 2022.

[4] Clara Marie Lüders, Mikko Raatikainen, Joaquim Motger, and Walid Maalej. Openreq issue link map: A tool to visualize issue links in jira. In *2019 IEEE 27th International Requirements Engineering Conference (RE)*, pages 492–493. IEEE, 2019.

[5] EH Miller. A note on reflector arrays (periodical style—accepted for publication). *IEEE trans. antennas propagat*, pages 123–135, 1990.

[6] Lloyd Montgomery, Clara Lüders, and Walid Maalej. An alternative issue tracking dataset of public jira repositories. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 73–77, 2022.

[7] Naresh Kumar Nagwani and Shrish Verma. A comparative study of bug classification algorithms. *International Journal of Software Engineering and Knowledge Engineering*, 24(01):111–138, 2014.

[8] Nitish Pandey, Debarshi Kumar Sanyal, Abir Hudait, and Amitava Sen. Automated classification of software issue reports using machine learning techniques: an empirical study. *Innovations in Systems and Software Engineering*, 13:279–297, 2017.

[9] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140):1–67, 2020.

[10] Timo Schick and Hinrich Schütze. It's not just size that matters: Small language models are also few-shot learners. *arXiv preprint arXiv:2009.07118*, 2020.

[11] Haike Xu, Zongyu Lin, Jing Zhou, Yanan Zheng, and Zhilin Yang. A universal discriminator for zero-shot generalization. *arXiv preprint arXiv:2211.08099*, 2022.

[12] Ruohong Zhang, Yau-Shian Wang, and Yiming Yang. Generation-driven contrastive self-training for zero-shot text classification with instruction-tuned gpt. *arXiv preprint arXiv:2304.11872*, 2023.

[13] [online] jira.atlassian.com