Early and Late Fusion Ensemble Methods for Predicting Bug Severity from Bug Report

by

Md. Farhan Farooq
20101083
MD. Shahariar Nawshad Nabil
20201191

A Final Thesis submitted to the Department of Computer Science and Engineering
in partial fulfillment of the requirements for the degree of
B.Sc. in Computer Science

Department of Computer Science and Engineering
Brac University
June 2025

Declaration

It is hereby declared that

1. The thesis submitted is my/our own original work while completing degree at Brac University.
2. The thesis does not contain material previously published or written by a third party, except where this is appropriately cited through full and accurate referencing.
3. The thesis does not contain material which has been accepted, or submitted, for any other degree or diploma at a university or other institution.
4. We have acknowledged all main sources of help.

Student's Full Name & Signature:

Md. Farhan Farooq	MD. Shahariar Nawshad Nabil
20101083	20201191

Approval

Approval

The thesis/project titled "Early and Late Fusion Ensemble Methods for Predicting Bug Severity from Bug Report" submitted by

1. Md. Farhan Farooq (20101083)
2. MD. Shahariar Nawshad Nabil (20201191)

of Spring, 2025 has been accepted as satisfactory in partial fulfillment of the requirement for the degree of B.Sc. in Computer Science on June 20, 2025

Examining Committee:

Supervisor:
(Member)

Md. Aquib Azmain

Lecturer
Department of Computer Science and Engineering
Brac University

Program Coordinator:
(Member)

Md. Golam Rabiul Alam, PhD

Professor
Department of Computer Science and Engineering
Brac University

Head of Department:
(Chair)

Dr. Sadia Hamid Kazi

Chairperson and Associate Professor
Department of Computer Science and Engineering
Brac University

Abstract

Accurate prediction of software bug severity is essential for optimizing resource allocation, enhancing bug triaging, and improving project management within the software development lifecycle. This study introduces a robust methodology for predicting bug severity by leveraging textual data from bug reports, employing advanced natural language processing (NLP) techniques and machine learning models. We evaluate several approaches, including Word2Vec with XGBoost (68% accuracy, 64% precision, 68% recall), TF-IDF with Logistic Regression/SVM (77% F1 score), DistilBERT (73% accuracy, 70% F1 score), and DistilRoBERTa (76% accuracy, 73% F1 score), each demonstrating strengths in capturing semantic and contextual nuances of bug descriptions. To further improve performance, we propose a fusion-based ensemble learning framework, combining early fusion (integrating TF-IDF, Word2Vec, and transformer embeddings into a unified feature vector) and late fusion (aggregating predictions from independently trained models). The hybrid Ensemble Fusion model achieves the highest performance, with an accuracy of 79% and an F1 score of 76%, excelling in generalizing across diverse bug severity, including challenging short and long durations. Our methodology encompasses rigorous data preprocessing, feature engineering, and techniques to mitigate class imbalance, utilizing a comprehensive dataset of bug reports with rich textual and metadata attributes. The results underscore the efficacy of integrating diverse feature representations and model predictions, providing a scalable, robust, and actionable solution for predicting bug severity, ultimately enhancing software development efficiency and reliability.

Keywords: Natural language processing (NLP), Machine learning, Bug reports, Early fusion, Late fusion, Ensemble fusion

Acknowledgement

Firstly, all praise to the Great Allah for whom our thesis have been completed without any major interruption.
Secondly, to our supervisor Aquib Azmain sir for his kind support and advice in our work. He helped us whenever we needed help.
I would also like to extend my thanks to my parents, whose boundless encouragement and prayers have been the driving force behind my academic accomplishments. This thesis would not have reached its completion without the collective support and belief of these individuals, for which I am deeply thankful.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Bug-Fixing Challenges

Creating software involves a series of steps that encompass designing, testing and maintaining a software application. In this long process, bugs or defects are unavoidable which creates numerous challenges to software developers and testers. If a bug is not treated right after its reported time, it can cause many problems from minor glitches to critical errors that impact the functionality and usability of a software. Besides, these bugs can be used to measure a software product's quality. According to [1], if a file contains 100 cumulative bugs during its development history, we may say that the file is more unstable than others without any bugs during the development period.

To maintain the quality and reliability of software products proper troubleshooting is a vital step. However, the process of this troubleshooting can be time-consuming and also resource-demanding. In [5] it is highlighted that If a software company spent one dollar on its product more than 30% of that dollar has gone to find and fix the bugs. This indicates bug troubleshooting is the most expensive event in SDLC. Moreover, research done by Cambridge University stated that developers spent half of their working time troubleshooting bugs In the era of paced software development, where fast releases and continuous development are common practices, it's quite important to tackle the defects quickly.

In this research, the primary objective is to develop a robust and accurate system for predicting the severity required to fix software bugs by leveraging textual data contained in bug reports. Software bugs are inevitable in the software development life cycle (SDLC) and have a significant impact on software quality, release timelines, and user satisfaction. Hence, timely identification and prioritization of bug fixes is critical for effective software maintenance and project management. This section introduces the overarching methodology adopted to achieve this objective by integrating natural language processing (NLP) techniques with advanced machine learning models, particularly focusing on fusion-based ensemble learning methods.

1.2 Bug-Severity Prediction

Predicting bug-fixing time is required to create effective project planning and proper use of allocated resources. Besides, good predictability helps software developers meet project deadlines, allocate resources efficiently, and focus on more severe bugs rather than wasting time on low-severity contained bugs. Moreover, in [13] it is mentioned that for excellent user experience and for perfect project planning fast bug-fixing time is a crucial part.

Previously, bug severity prediction was highly dependent on historical data and statistical models. These approaches were able to provide valuable information from past bug resolution patterns but often failed to gather rich textual information from bug reports. Also, most of these methods were unable to show the correlation between different attributes of the dataset. For example, In [8] authors used multivariate and univariate regression testing to predict the bug severity and to analyze bug report attributes correlation but they found bug severity and bug-opener's reputation has no relation between them. A detailed description of the reported bugs, information on the system where the bug was encountered, and steps to face the mentioned bug, especially this data are mentioned in the bug report. If the gathered data is analyzed properly it can provide valuable information like the bug's nature, severity, and complexity, which can be helpful to predict bug severity more accurately. For example, the proposed method in [13] was able to obtain the weighted F-measure from 65 to 85, within an average of 72.45. It was possible because for predicting bug-fixing time the most important features were the reporter and owner attributes. So, textual data can be an important factor in predicting the troubleshooting time.

The most consuming and vital stage in SDLC is usually at the bug fixing stage. Studies have shown that developers are to a large extent engaged in the processes of comprehending, allocating priorities, and correcting defects, which have a direct impact on the productivity and hardware reliability. Productive resource planning, schedule improvement, and bug triaging, and efficiency of developers are some of the advantages of being able to know the waiting time you would expect a bug to be fixed. The traditional approaches have been based on historical numerical data or metadata but underestimate the high textual data content of the bug reports that is also a vital source of information on the complexity and nature of bugs.

Bugs are in general represented by several text filled in fields including a brief summary, extensive description of bug, software affected by bug, instructions to replicate a bug, and/or developer feedback. These textual fields contain intricate data such as the severity of the bug, its background in the software system and also the challenges that may be associated with repairing the bug. The tools required to extract the meaningful semantic features using this unstructured text data are offered by the so-called natural language processing (NLP) techniques. One can achieve higher accuracy and reliability in predicting severity than the purely numerical or metadata-driven models can by transforming the textual data so that numerical representations become available to a machine learning algorithm.

1.3 Research Problem

Since the inception of software development in history bugs have remained a very unavoidable aspect of the process. Since the beginning of programming, up to the era of the software systems developers have been presented with the responsibility of identifying and resolving bugs to ensure the high level and reliability of their products. However, as improvements are made in the areas of software engineering techniques and materials, beating bugs to the time of its occurrence remains a priority, on the part of software development teams.

In the bug resolution process, one of the key challenges is the accurate prediction of the time required to fix a bug. The uncertainty surrounding bug fix durations can result in software release delays, project deadline misses, and higher expenses. Traditional approaches to bug severity prediction were heavily dependent on historical data and statistical models. For this reason, the rich textual information contained within bug reports was often neglected. This drawback impacts prediction precision and hampers bug resolution procedures.

Moreover the rising intricacy of software systems and the surge, in bug reports intensify the difficulty in predicting bug fixing time. Development teams face a multitude of reported bugs from users and testers each presenting features and impacts on the software's performance. In the absence of prediction tools and methods for estimating fix times developers might find it challenging to manage resources and prioritize tasks which could result in inefficiencies, in resolving bugs and potentially compromise the quality of the software.

1.4 Overview of the Proposed Methodology

We are approaching with a new method in this paper to predict bug severity by imposing NLP techniques on bug reports. We are using a bug report dataset from Free/Libre Open Source Software (FLOSS) projects[25]. From this dataset we are aiming to extract relevant features. By using NLP techniques we are hoping to analyze linguistic patterns and semantic information contained within the report to gain a deeper understanding of the reported issue. After that, we will use different machine-learning algorithms which will help us to predict bug severity more accurately. Through carrying out tests, on bug report datasets we showcase the effectiveness of our method. Specifically, our research objectives are as follows:

1. Data Collection and Understanding: Utilizing a well-established dataset of bug reports from Free/Libre Open Source Software (FLOSS) projects, which contains long and short textual descriptions, bug severity labels, and severity.

2. Data Preprocessing and Feature Engineering: Applying NLP preprocessing steps such as tokenization, stopword removal, lemmatization, and vectorization to convert textual data into machine-readable formats. Additionally, metadata features such as severity codes are processed for use in models.

3. Model Development: Implementing multiple machine learning models with different text representation techniques to extract features from bug descrip-

tions. These include traditional vectorizers like TF-IDF and Word2Vec as well as transformer-based contextual embeddings such as DistilBERT and DistilRoBERTa.

4. Fusion-based Ensemble Learning: Integrating multiple models through early and late fusion strategies to leverage complementary strengths of different text representations and classifiers, improving overall predictive performance.

5. Model Evaluation and Analysis: Using standard classification metrics such as accuracy, precision, recall, and F1-score to rigorously evaluate model performance and interpretability.

By meeting these goals our aim is to help improve the bug-fixing procedures, in software development projects. Our study aims to provide software development teams with the tools and methods to predict bug resolution time precisely. This will assist them in allocating resources, meeting project timelines, and delivering top-notch software products to users.

1.5 Detailed Explanation of Methodology Components

1.5.1 Data Preprocessing:

Given the unstructured nature of bug report descriptions, preprocessing is a crucial step to clean and prepare the text for modeling. This involves:

- **Tokenization:** Breaking down text into individual words or tokens.
- **Stopword Removal:** Eliminating commonly used words that carry little predictive meaning, such as "the," "and," or "is."
- **Lemmatization:** Reducing words to their base or root form to unify different morphological variants (e.g., "fixing," "fixed," "fixes" $\rightarrow$ "fix").
- **Vectorization:** Transforming processed text into numerical vectors using TF-IDF, Word2Vec embeddings, or transformer-based embeddings.

1.5.2 Text Representation Models:

- **TF-IDF (Term Frequency-Inverse Document Frequency):** A classic method assigning weights to words based on their frequency in a document relative to their frequency across all documents. This highlights important words unique to particular bug reports.
- **Word2Vec:** A neural network-based approach generating dense word embeddings that capture semantic relationships between words. Averaging these embeddings creates a feature vector representing the bug description.

- **Transformer-based Models (DistilBERT, DistilRoBERTa):** Advanced pre-trained language models that generate context-aware embeddings, capturing nuanced meanings of words depending on their surrounding text. These models have demonstrated state-of-the-art performance in many NLP tasks, especially with domain-specific fine-tuning.

This pipeline ensures that models receive clean, relevant, and compact textual data, enabling efficient learning.

1.5.3 Fusion-based Ensemble Learning:

While individual models possess unique advantages, combining their predictions can often improve robustness and accuracy. Two main fusion techniques are employed:

- **Early Fusion:** Combining feature vectors from multiple models into a single concatenated feature representation before feeding them into a classifier. This approach allows the model to learn joint representations from diverse inputs.
- **Late Fusion:** Independently training separate classifiers on different feature sets and combining their outputs (e.g., via majority voting or weighted averaging). This allows leveraging diverse model decisions for final prediction.

Using both strategies ensures that the system benefits from both feature-level integration and decision-level consensus, enhancing performance especially when dealing with complex and noisy bug data. This methodological framework integrates advanced NLP techniques and ensemble learning approaches to provide a powerful system for predicting bug severity from unstructured bug reports. By combining multiple models and leveraging rich textual information, the approach aims to deliver highly accurate and actionable predictions that can greatly assist software teams in optimizing bug triage and resource allocation. The methodology is designed to be scalable, adaptable, and practical, aligning with real-world software development needs.

Chapter 2

Literature Review

To ensure product quality and user satisfaction the efficient management of software bugs has become a highlighted aspect in the era of the progressed software development industry. It is essential to identify and resolve software bugs on time to ensure competitive advantage and produce dependable goods. To improve bug severity prediction, the use of predictive analytics techniques—specifically, machine learning (ML) and natural language processing (NLP) algorithms has become a promising approach.

2.1 Predictive Analytics in Bug Fixing

Bug fixing is a crucial process in the software development life cycle and must be conducted judiciously and timely to maintain high-quality products that keep the users satisfied. Bug severity prediction can be improved with predictive analytics techniques such as natural language processing and machine learning algorithms. Notably, various studies have confirmed that bug severity prediction results in early bug identification, reduced development costs, reduced project delays, and resource allocation.[7], [10]

Recently, some works have been done on this which enables the academic to predict the duration of bugs to be get fixed more accurately by extracting necessary information from textual bug reports. Classification machine learning algorithms like Decision Trees, Random Forests, and Gradient Boosting Machines, are being used to analyze bug report data and identify key factors influencing bug fix time.[9], [12] In a study by Gao et al. [9] extracted the right feature and used Random Forest to predict time and got will over 80% accuracy.

2.2 Integration of NLP Techniques

By collecting semantic data and linguistic patterns from issue reports, the inclusion of NLP techniques improves the forecast of bug severity even further. Natural language processing (NLP) models can detect correlations by analyzing different attributes such as bug severity, description, and affected components.[7], [22] Furthermore, deep learning architectures provide strong frameworks for capturing intricate

correlations in bug report data, such as Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks.[9], [10]

Many studies have proven the effectiveness of NLP-based bug severity prediction models in real-world scenarios, by showing their ability to precisely predict bug severity and prioritize bug resolution tasks. For example, Mani et al. [22] developed an NLP-based model that surpassed all the traditional machine-learning approaches to predict bug severity for the Mozilla and Eclipse projects. Moreover, software development teams can speed up bug-triaging processes and effectively allocate resources by harnessing the power of NLP techniques by which product quality and project efficiency will increase.[12], [22]

2.3 Detailed Literature Review

Over the years, researchers have developed many techniques to help estimate how much time it will take to resolve software errors. Accurately estimating bug fix duration is necessary for effective handling of projects, arranging resources and providing good software on schedule. This section includes a detailed overview of previous studies in this subject, mentioning what the researchers wanted to discover, the data they analysed, the tools they used, the results they found and any challenges they experienced. By studying each piece of research, we want to put the current research into context and identify the main successes and problems in the field.

2.3.1 Bug Severity Analysis in Industry-Level Software

The research by [1] studied the amount of time needed tofix bugs in large industry-level software systems, using ArgoUML and PostgreSQL as examples. Most people think of these software systems because they have highly complex designs and a lot of users. The main goal of the study was to know how long the bugs stayed every time and to locate files that were usually more difficult to address. To find what could be better in the design, the authors looked at these patterns and decided what needed to be improved to fix bugs and make the software maintainable. The process focused on watching the journey of bugs, starting with their discovery and going right through to the finish line and then arranging the top 20 files by the time it took to correct their bug. Suggestions were given to consider re-engineering the files so they would help improve the software.
While the results showed areas where bugs were likely to happen, the predictive accuracy was only about 60%. Still, the authors acknowledged it was possible that their approach missed parts of the problem because it didn't analyse code complexity, experience levels or how effectively teams collaborate. Besides, this study only looked at two projects which brings into question if the results can be applied to a broader audience. A wider sample of data could have given a better sense of how bug fix times differ between different software programmes.

2.3.2 Predicting Bug Severity Using Data Mining Tools on Textual Data

In the study [2], authors focused on whether bug fix prediction is possible simply by analysing text from bug reports right after they are created. With use of the Eclipse Bugzilla data, the authors created models that estimate how long it takes bugs to be fixed. They relied on the rollback technique to show what happens in situations with limited knowledge when the bug report is made, like in actual practise.
Even though rollback techniques were key to the study, only 34.9% of predictions could be made correctly. Since text does not usually contain such details as developer skills, how the system is built and team processes, even expert programmers can encounter problems fixing them. They mentioned that their model was missing higher-level characteristics like bug severity and how much priority the project team gave it which might have improved how well it predicted dates. What's more, since the study looked only at new reports, its findings weren't useful for handling all types of bug reports, as it did not include earlier and connected bug data. It creates opportunities for future research to combine better datasets and more elaborate modelling approaches.

2.3.3 Early Bug Fix Prediction Through Issue Tracker Data and Textual Matching

The research by [4] investigated early prediction of bug fix times by analyzing issue tracker data and applying text-matching techniques. Using the JBoss Project database, the authors employed k-Nearest Neighbors (kNN) and adaptive k-Nearest Neighbors (a-kNN) algorithms to predict the amount of effort required to resolve a bug. They validated their predictions by cross-referencing them against historical data, showcasing the potential of using similarity-based methods to estimate fix times.
The researcher discovered that the text similarity methods executed better in the prediction of accuracy compared to naive baselines. However, the limitations of the corresponding study cannot be neglected. The results are not generalizable to other systems since the project work just concentrated on the JBoss project only. The model was based on issue reports with their focus on the explicit data of efforts, which could exclude a considerable part of reports and provide the bias of results to the particular types of anomalies (bugs). This has been rather too narrow perhaps resulting to bias by not taking into consideration the complexity and diversity behind real-world bug situations. However, the research assigned the significance of text data in predicting times of bug fix and laid foundations of further investigation of similarity-based methods.

2.3.4 Bug Fix Time Prediction Models in Software Development

The work of [6] contributed to the development of decision tree-based models of bug fix prediction in software development projects. Based on various data sets, such as bug reports on Eclipse, Mozilla and Gnome projects the authors have implemented decision tree algorithms in concert with the 10-fold cross-validation method of eval-

uating their models. The research work recorded impressive outcomes, where the precision recall ratio was close to 0.75 and Area under the Curve (AUC) score was at 0.834.
Although encouraging results were obtained, the study had some limitations. The authors took into consideration that their models did not take into account such external factors as team dynamics and individual developer experience. These issues are capable of drastically affecting the time that it takes to rectify the bugs particularly within the collaborative and large scale environments. Nonetheless, the work executed allows concluding that decision trees are effective in the prediction of bug fix time and highlights the importance of textual information combining with classic machine learning models.

2.3.5 Prioritizing Bug Fixes Based on Bug Report Edits and Personnel

The work of [7] was concerned with the bug priorities in repairing bugs depending on numerous aspects that included the content of the bug reports as well as the responsibilities of the people involved in the repairing of bugs. Qualities based on that are obtained using the Windows vista bug database (dataset of bug reports, surveys of Microsoft workers were conducted). These characteristics were used to develop logistic regression models which tried to predict the precedence of bugs.
The experimental, however, obtained the accuracy of 68%, which indicates the potential of feature engineering and logistic regressions as the means of prioritizing bug fixes. The constraints however involved elimination of the priority field given discrepancies in the ratings of the bug importance, and the fact that bug type did not make great difference in terms of the model precision. Such constraints imply the necessity to develop more effective feature selection and putting more contextual factors into consideration in the future research. The study however gave good points in the connection of bug report properties, developer relations, and fix duration.

2.3.6 Large-Scale Regression Testing for Bug Fix Time Prediction

A comprehensive study by [8] conducted regression analysis on an extensive dataset comprising 512,474 bug reports from projects like Eclipse, Chrome, Firefox, Mozilla, Seamonkey, and Thunderbird. The authors applied multivariate and univariate regression techniques to predict bug fix times and assess the predictive significance of various attributes.
Their findings revealed that existing models exhibited predictive powers between 30% and 49%, indicating moderate success. A key observation was the lack of correlation between bug fix time and the bug-opener's reputation or bug severity, challenging assumptions that experienced reporters or severe bugs necessarily influence fix times. While the study's breadth was impressive, its depth was limited by the absence of feature validation across different project scales, and the results suggested a need for more sophisticated models capable of capturing nuanced relationships within bug data.

2.3.7 Studying Bug Fix Time Across Multiple Dimensions

The research by [9] examined bug fix times along multiple dimensions using bug report data from Eclipse and Mozilla projects. Employing a Random Forest classifier, the authors analyzed bug fix durations and achieved an accuracy of 65%. The study's multi-dimensional approach offered a broader perspective on factors influencing bug fix times.
However, limitations included a focus solely on these two projects, which may not generalize across different software systems. The authors also highlighted the need for further studies on other types of systems to validate their findings. Despite these challenges, the study demonstrated the potential of ensemble models in improving bug fix time prediction.

2.3.8 Filtering and Enhancing Bug Fix Time Prediction

In [11], the authors added a filtering stage in order to improve the quality of bug fix time forecasts. They also used sophisticated methods to detect outliers in bug reports like Chauvenet Criterion, and used the data mining methods to predict. Their results implied that prediction accuracy was helped by filtering of selective bugs and elimination of outliers especially when there was a rapid correction of the bug. Nonetheless, other limitations, including the data skew in terms of the fix time (some reports marked it as resolved within minutes) and the lack of integrating the reopened bugs that may bias the results due to selecting a particular set, were also observed in the research. The importance of preprocessing (cleanning the data) in bug fix time was emphasised in the study.

2.3.9 Monte Carlo Simulation for Predicting Fix Times

The article by[13] aimed at estimating fix times with the help of the Monte Carlo simulation and a method based on the Markov approach in commercial-level programs. The researchers relied on bug datasets of the CA Technologies projects as well as classification models that were implemented to foretell whether there was a slow bug fixing or quick bug fixing, and feature ranking algorithms, including Chi-Square, Gain Ratio, and Information Gain.
The results varied across projects, with the Mean Relative Error (MRE) for Project A ranging from 2.831 to 11.686, Project B between 1.845 and 4.633, and Project C between 0.015 and 1.070. While these findings demonstrated the utility of simulation and probabilistic methods, limitations included small datasets and occasional inaccuracies in fix time predictions, underscoring the need for more robust validation.

2.3.10 Predicting Bug Fix Time with Association Rule Mining and Clustering

The study by [14] explored bug fix time prediction using Apriori algorithms and k-means clustering on bug reports from Bugzilla. The authors aimed to predict the time taken to fix bugs and also to identify clusters of related variables, thus providing a dual perspective on the bug-fixing process. Their approach revealed that bug fix time prediction accuracy ranged between 21% and 100%, depending on the

specific bug category, while defect association prediction achieved an accuracy of 95.38%.
Although the study's use of association rules provided valuable insights into how certain attributes might correlate, the authors acknowledged limitations. The study was confined to open-source Mozilla products, potentially restricting its applicability to other software projects, particularly commercial systems. Furthermore, the authors did not fully address errors in performance measures like accuracy, leaving some uncertainty about the model's reliability across diverse contexts. Nevertheless, the study demonstrated the potential of combining clustering and association rule techniques in understanding bug dynamics.

2.3.11 Mapping Bug Fix Time Estimation as a Text Categorization Problem

In a novel approach, [15] redefined bug fix time estimation as a text categorization problem by leveraging all available textual information, rather than selecting specific features. Using datasets from multiple open-source projects such as OpenOffice, Eclipse, Gentoo, and KDE, the authors employed Supervised Latent Dirichlet Allocation (SLDA) models combined with bag-of-words and Vector Space (VS) representations to predict bug fix times.
The study reported varying accuracies across projects: for instance, OpenOffice with SLDA K=10 achieved an accuracy of 0.51, Eclipse with K=25 reached 0.57, Gentoo with K=30 attained 0.43, and KDE with K=40 recorded 0.41. These results, while moderate, underscored the potential of treating bug fix time prediction as a text categorization task. Limitations included the inability of defect tracking systems to capture actual person-hours spent on bug fixes, as well as challenges in identifying and filtering outliers within bug fix time distributions. These factors highlighted the need for integrating richer, more structured data with textual analysis for improved performance.

2.3.12 Predicting Bug-Fixing Effort Through Code Churn Analysis

The study by [16] approached bug-fixing effort prediction by focusing on code churn—the amount of code changed to resolve a bug. Using 1,029 bug reports from hadoop-common and struts2 Apache projects, the authors employed a classification-based model to estimate the effort required.
It had an Area Under the Curve (AUC) rate of 0.612, that is, 22.4 percent of an additional rate in respect to baselines. Yet the authors admitted that there are no automated approaches which are created to predict the size of code churn directly, and the prediction of fix times is not always correct. This puts in perspective, there is a gap in closing the association between code churn and the real bug fixing times. Although the paper has narrowed down the usefulness of code changes to interpret bugs-fixing activities, another issue that remains relevant is that we pay more attention to holistic models, that take textual issues as well as code-based characteristics, into consideration.

2.3.13 Predicting Bug Fix Time Using SVM Classifiers and Textual Information

In the article by [17] Support Vector Machine (SVM) classifiers were used in prediction of bug-fix time, using bug reports of Novell, OpenOffice and LiveCode projects. The authors intend to predict the amount of time it would take to solve the bug through text mining analysis of bug descriptions.
They gave different accuracy scores: 0.35 with LiveCode and 0.52 with OpenOffice, which implies that they were unable to generalize their solutions in various projects with ease. Among the limitations were the fact that there might have been problems in the choice of field to estimate bug-fix times, and that the model was not applicable to non-open source projects extensively and that there might have been some outliers in the data. Although this was the case, the research demonstrated that SVM classifiers might lead to the development of a suitable basis upon which bug fix time prediction might be based when textual information is incorporated. Even though it is not quite accurate yet, it still shows some promise.

2.3.14 Predicting Bug Fix Time Using Hidden Markov Models

In [18], they proposed a time-series approach based on an application of Hidden Markov Models (HMMs) to predict the time it takes to fix a bug where the bug reports were obtained in a Firefox project. They made developer activity temporal sequences part of their model and were able to predict fix durations 70.73 percent accurately. Though encouraging, the study experienced potential reports of overfitting the HMMs, as well as the observed threats of a threat to conclusion validity because of a lack of bug report activities. It was constrained by the lack of actual real-time developer efforts. Notwithstanding these limitations, the study identified the usefulness of temporal modeling toward exploring the bug-fixing workflow and provided potential directions in how more granular data about developer activities can be incorporated in a study in the future.

2.3.15 Forecasting Monthly Bug Fix Counts and Cumulative Fix Time

The work of [19] implemented bug prediction to the next level by predicting how many bugs a month consisted of, which could be solved, and how much cumulative time it will take to complete bug fixing. The authors used a combination of Markov models in the form of prediction of the number of bugs that get fixed monthly, and Monte Carlo simulations of fix times in terms of cumulative, as well as k-Nearest Neighbors (kNN) to classify fix times using Bugzilla datasets.
They found that similarities existed in both the bug management practices among various systems whereas their findings also indicated limitations which included the possibility of biases in sampling procedures in kNN and the non-applicability of Markov models to projects with limited life spans of the bug. This paper demonstrated the possibilities of building probabilistic models and simulations methods, through which we may obtain more general understanding of the bug-resolving trends over the time.

2.3.16 Probabilistic Model-Based Approaches for Automated Program Repair (APR)

A study by [21] considered a probabilistic-based APR method where the optimal set of strategies to implement bug fixes is selected; a subset of Defects4j benchmark data set was used. They created a two-level probabilistic model of the operator selection and instantiation in problem-solving operation during the process of bug fix, with an onset of 63% accuracy.
However, the study required manual input for identifying faulty locations, limiting its automation potential. There were also concerns about the generalizability of generated patches to different program behaviors, as low-quality patches might not transfer well across diverse scenarios. Despite these challenges, the study contributed insights into leveraging probabilistic models for bug-fixing strategy selection, suggesting potential for further automation with improved techniques.

2.3.17 Predicting Bug Fix Time and Severity Using Multiple Predictive Models

The study by [23] employed various predictive models—including SVM, Naive Bayes, and k-Nearest Neighbors—to estimate bug fix time and severity, using bug reports from Mozilla open-source software products. The study also integrated text mining techniques to analyze bug summary weights and derive bug age features. For bug fix time prediction, they utilized Support Vector Regression (SVR) and k-Nearest Neighbors Regression.
Accuracy across various bug attributes ranged from 37.34% to 91.23%, though the authors noted the absence of grid search for optimal parameter tuning. Additionally, the study did not address reopened bugs or handle bug fix times separately, which could affect the model's precision. Nevertheless, this study underscored the value of combining multiple predictive models and textual analysis for comprehensive bug fix time estimation.

2.3.18 Deep Learning and BERT for Bug Fix Time Prediction

[24] made use of deep learning by adopting BERT methods to predict the time to fix bugs which was one of the latest studies reviewed here. Authors analysed the description of each issue together with the comments from developers in reports from both Bugzilla and LiveCode. They used transfer learning, adapting pre-trained BERT models to predict how much time will be needed to fix a bug. Different benchmarking investigations also compared decision tree and SVM models.
Precision of the decision tree was calculated at 0.654, with recall at 0.692 and SVM achieved an accuracy of 0.77. But the study had some problems because it didn't use hyperparameter tuning, so model performance could have been improved and because the Bugzilla data about developer effort was not considered, resulting in less accurate predictions. Even so, the study points out that BERT and similar advanced NLP models have potential for bug fix time prediction and that improvements can be made by optimising their parameters and including more detailed information in models.

2.3.19 Discussion

The reviewed papers show a continuous progress in the approaches and models that predict bug severity. Various studies exploring different techniques and models such as regression, decision trees, ensemble learning and BERT, have helped build a better understanding of what affects the time required to resolve a bug. Common problems in studies are that the datasets are limited, hyperparameters are seldom tuned, developer effort and team conditions are not included and models find it difficult to be applied in different kinds of software projects. While facing these difficulties, using text, code features, developer activity and probabilistic models has set a firm base for future steps in this field. According to the literature, combining different types of data—text, structure and behaviour—in bug severity prediction models helps ensure the accuracy and reliability of the forecast.

Chapter 3

Methodology

The workflow describes the step-by-step procedures followed to transform raw bug report data into accurate predictions of bug severity. It integrates natural language processing techniques, multiple machine learning models, and ensemble fusion methods to build a robust predictive system. This section details each stage of the workflow, from initial input processing to final bug severity classification, highlighting key decisions and methodologies employed throughout the process.

3.1 Overview

At a high level, the algorithm workflow consists of the following major stages:

- **Input Acquisition:** Collection of raw bug reports including textual descriptions and metadata.
- **Text Preprocessing:** Cleaning, tokenization, and normalization of text data.
- **Feature Extraction:** Transformation of text into numerical representations through different embedding techniques.
- **Model Training:** Training multiple predictive models based on different feature sets.
- **Fusion and Ensemble Learning:** Combining predictions or features from individual models to improve accuracy.
- **Prediction and Evaluation:** Producing bug severity predictions and assessing model performance.

These steps work sequentially and iteratively to ensure optimal model performance and robustness.

3.2 Input Acquisition

The workflow begins with obtaining the bug reports dataset, which includes:

- **Long Description:** Detailed textual explanation of the bug.

- **Short Description:** Concise summary of the issue.
- **Severity Codes:** Numeric or categorical labels indicating bug severity.
- **Fix Time Labels:** Target variable indicating time to fix the bug.
- **Additional Metadata:** Such as number of comments.

These inputs form the raw data from which meaningful features will be extracted.

3.3 Text Preprocessing

Given that bug descriptions are unstructured text, preprocessing is crucial to reduce noise and standardize data for modeling. The preprocessing pipeline includes:

- **Lowercasing:** Convert all text to lowercase to maintain consistency.
- **Removing Special Characters:** Strip punctuation, URLs, and non-alphanumeric characters that do not contribute to semantic understanding.
- **Stopword Removal:** Exclude common words with little discriminative value.
- **Tokenization:** Splitting text into tokens (words or subwords).
- **Lemmatization:** Converting words to their root forms to unify similar words.
- **Handling Domain-specific Terminology:** Retain or standardize key software engineering terms.
- **Padding and Truncation:** For transformer models, sequences are padded or truncated to a fixed length to meet model input requirements.

The result is clean, tokenized text ready for embedding generation.

3.4 Feature Extraction

To effectively capture the semantic and syntactic nuances in bug descriptions, the workflow employs multiple feature extraction techniques:

- **TF-IDF Vectorization:** Represents text by the importance of words relative to the corpus. It provides a sparse, high-dimensional representation emphasizing unique and informative words.
- **Word2Vec Embeddings:** Pre-trained or custom-trained word vectors that encode semantic relationships between words in dense numerical vectors. Averaged to obtain a fixed-length feature vector for each bug report.
- **Transformer-based Embeddings (DistilBERT, DistilRoBERTa):** Generate context-aware dense embeddings that reflect word meanings within the sentence context, capturing long-term dependencies and subtle semantic differences.

By extracting features through these diverse methods, the workflow obtains complementary perspectives on the textual data, strengthening the predictive power of subsequent models.

3.5 Model Training

Following feature extraction, multiple machine learning classifiers are trained independently using different feature representations:

- **Logistic Regression and Support Vector Machine (SVM):** Trained using TF-IDF features. These linear classifiers are efficient for high-dimensional sparse data and provide baseline performance.
- **XGBoost (Extreme Gradient Boosting):** Trained using averaged Word2Vec embeddings. XGBoost is a powerful gradient-boosting decision tree algorithm that captures nonlinear relationships and interactions.
- **Transformer Fine-tuning:** DistilBERT and DistilRoBERTa models are fine-tuned end-to-end on the bug severity classification task, leveraging their pre-trained knowledge to capture complex contextual dependencies.

Hyperparameter tuning is performed for all models using cross-validation and grid search to optimize performance.

3.6 Fusion and Ensemble Learning

The workflow incorporates fusion strategies to leverage the strengths of individual models and feature representations:

- **Early Fusion:** Combines feature vectors from TF-IDF, Word2Vec, and transformer embeddings into a unified representation. This concatenated vector is fed into an ensemble classifier (e.g., XGBoost or neural network) trained to predict severity. Early fusion allows the model to learn integrated feature interactions.
- **Late Fusion:** Independently trained classifiers generate prediction probabilities or class labels, which are combined using techniques such as majority voting, weighted averaging, or stacking. This decision-level fusion balances individual model biases and improves robustness.

By employing both early and late fusion approaches, the workflow achieves improved generalization and predictive accuracy compared to any single model.

3.7 Prediction and Output

For new bug reports, the trained models follow the same preprocessing and feature extraction steps, producing predictions on bug severity categories. The system outputs:

- **Confidence Scores:** Probability estimates or confidence values for each predicted class, useful for prioritization.
- **Explanatory Features (optional):** Feature importance or attention maps that provide insights into what textual elements influenced the prediction.

The algorithm workflow integrates sophisticated text preprocessing, diverse feature extraction methods, and multiple machine learning models combined through fusion techniques. This layered approach is designed to exploit the rich semantic information in bug reports, address the challenges of noisy and imbalanced data, and deliver reliable predictions of bug severity. The workflow is modular, enabling future extensions such as incorporating additional metadata or adopting novel fusion strategies.
-
We have used multiple models in our research. Our main goal was to estimate the time it takes to fix bugs based on the descriptions, in bug reports.

3.8 Word2Vec and XGBoost

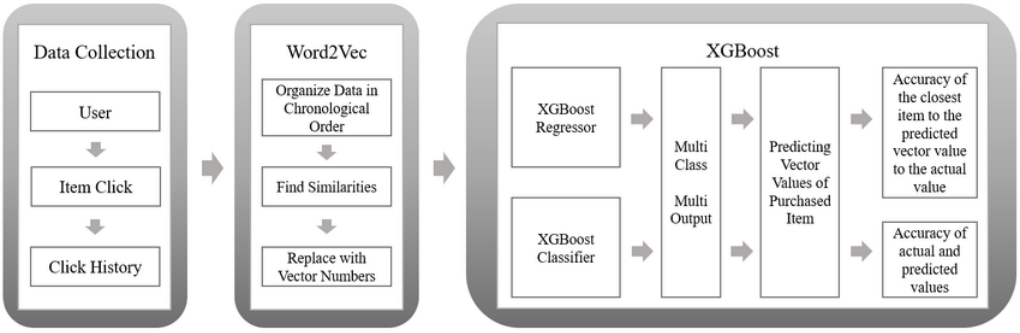

Figure 3.1: [26]Working architecture of Word2Vec and XGBoost.

Initially we used Word Embedding (Word2vec) in combination, with XGBoost In this process. Word2Vec converts bug descriptions, into multifaceted vectors so as to capture the underlying semantics among words. These vector representations of words are then averaged to generate a feature vector that encapsulates each individual bug report. Let $T = \{w_1, w_2, \ldots, w_n\}$ be a bug report containing n words.
Each word w_i is mapped to a d-dimensional vector using Word2Vec:

$$\phi(w_i) \in \mathbb{R}^d$$

We average the word vectors to get a report-level feature vector $\mathbf{x}$:

$$\mathbf{x} = \frac{1}{n} \sum_{i=1}^{n} \phi(w_i)$$

This resultant feature vector $\mathbf{x}$ is subsequently inputted into XGBoost classifier:

$$\hat{y} = f_{\text{XGB}}(\mathbf{x})$$

where f_{XGB} is an ensemble of decision trees trained to minimize a regularized objective:

$$\mathcal{L} = \sum_{i=1}^{N} \ell(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k)$$

$$\text{where } \Omega(f) = \gamma T + \frac{1}{2}\lambda \|\mathbf{w}\|^2$$

ℓ: loss function (e.g., log loss for classification)

f_k: the k-th regression tree

T: number of leaves in the tree

$\mathbf{w}$: leaf weights

γ, λ: regularization parameters

XGBoost constructs a set of decision trees for predictive analysis and we conducted thorough fine tuning of hyperparameters for optimal performance.

3.9 TF-IDF and Logistic Regression

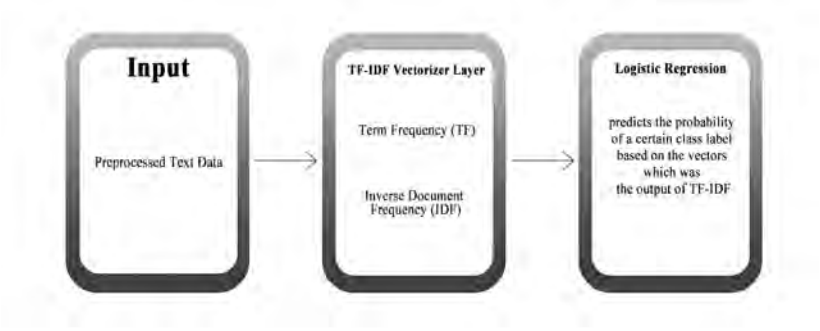

Figure 3.2: Model architecture of TF-IDF and Logistic Regression.

We have also implemented TF-IDF (Term Frequency-Inverse Document Frequency). It mainly transforms textual data of bug descriptions to numerical vectors focusing important terms of the bug description. TF-IDF mainly focuses on the important words of the description while ignoring the less informative recurrent words by assigning them lower numeric values. The Logistic Regression and SVM models were then fine tuned using these vectors for making predictions about bug severity. TF-IDF was used to vectorize bug report texts by highlighting terms with discriminative importance, and Logistic Regression was applied to classify the severity categories. For a word t in document d, and corpus D:

$$\text{TF-IDF}(t, d, D) = \text{tf}(t, d) \cdot \log\left(\frac{|D|}{1 + \text{df}(t)}\right)$$

Let $\mathbf{x} \in \mathbb{R}^n$ be the resulting sparse TF-IDF vector of a bug report.

Logistic Regression computes the probability that a report belongs to severity class $y \in \{0, 1\}$:

$$P(y = 1 \mid \mathbf{x}) = \sigma(\mathbf{w}^\top \mathbf{x} + b) = \frac{1}{1 + e^{-(\mathbf{w}^\top \mathbf{x} + b)}}$$

For multi-class severity prediction, softmax is used:

$$P(y = c \mid \mathbf{x}) = \frac{e^{\mathbf{w}_c^\top \mathbf{x} + b_c}}{\sum_j e^{\mathbf{w}_j^\top \mathbf{x} + b_j}}, \quad c = 1, \ldots, K$$

The model is trained by minimizing cross-entropy loss:

$$\mathcal{L} = -\sum_{i=1}^{N}\sum_{c=1}^{K} y_{i,c} \log P(y_i = c \mid \mathbf{x}_i)$$

These linear classifiers have proven proficient in assignments though they might encounter challenges, with intricate contextual comprehension.

3.10 DistilBERT and DistilRoBERTa

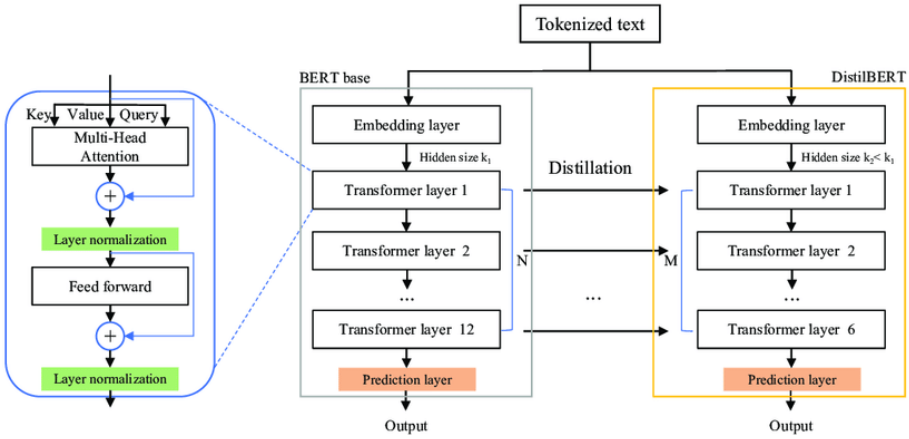

Figure 3.3: [27]Model architecture of DistilBERT.

We used transformer based models called DistilBERT and DistilRoBERTa. Which falls under the BERT based architecture group in which a pre trained transformer is tailored to bug report data refining their understanding of word and sentence interactions, within the text content to generate embeddings. The obtained embeddings are then fed into a classifier to forecast bug severity. DistilBERT and DistilRoBERTa stand out at comprehending lengthy. Intricate bug reports, by capturing context across sentences which surpasses the capabilities of traditional methods. Given an input text sequence $T = \{w_1, w_2, \ldots, w_n\}$, tokenized as $X = [x_1, x_2, \ldots, x_n]$, the transformer model computes contextual embeddings through stacked self-attention layers.

For each token x_i, the representation is:

$$h_i^{(l)} = \text{TransformerBlock}^{(l)}(h_i^{(l-1)}), \quad h_i^{(0)} = \text{Embed}(x_i)$$

The final hidden state of the [CLS] token (or average pooling of token embeddings) is used as the report embedding $\mathbf{z} \in \mathbb{R}^d$:

$$\mathbf{z} = \text{BERT}_{\text{CLS}}(X)$$

This embedding is passed through a classification head:

$$\hat{y} = \text{softmax}(W\mathbf{z} + b)$$

The model is trained via categorical cross-entropy:

$$\mathcal{L} = -\sum_{i=1}^{N}\sum_{c=1}^{K} y_{i,c} \log \hat{y}_{i,c}$$

Where:

$\mathbf{z}$: contextual embedding from transformer

W, b: trainable weights

$\hat{y}$: predicted probability distribution over severity classes

This design of their transformer structure enables them to sustain a grasp throughout reports and leverages transfer learning by pre training, on extensive collections of general text data. This blend of comprehension and pre existing knowledge renders them notably efficient, in scrutinizing bug reports and projecting resolution durations.

3.11 Early Fusion

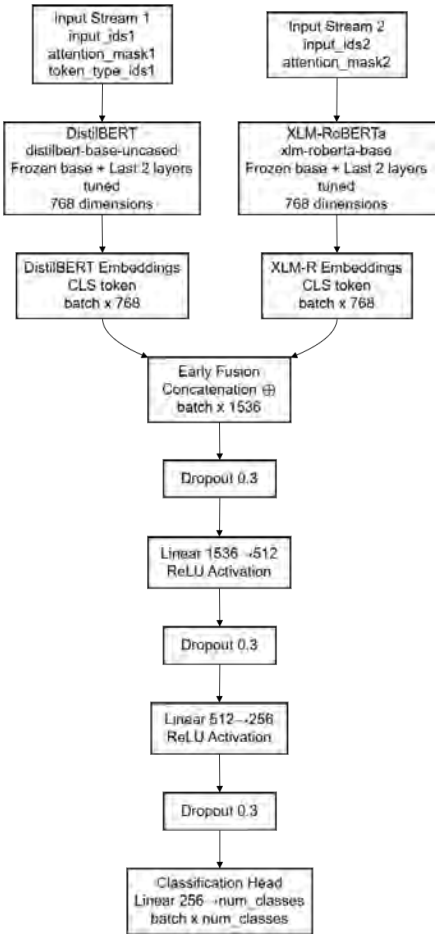

Figure 3.4: Model architecture of Early Fusion.

Early fusion is a feature-level integration technique that combines multiple distinct feature representations by concatenating them into a single, unified vector before feeding the data into a machine learning model. This approach leverages the complementary nature of different feature extraction methods, enabling the predictive model to learn joint interactions between heterogeneous features simultaneously. For the task of predicting bug severity from textual bug reports, early fusion integrates TF-IDF vectors (capturing term frequency importance), Word2Vec embeddings (capturing semantic relationships between words), and transformer-based embeddings like those from DistilRoBERTa (capturing rich contextual information) into a comprehensive representation. This combined feature vector allows the model to better understand the complexity and nuances of bug descriptions, ultimately improving the accuracy of severity predictions. Mathematically, assume the bug report is represented by K different feature vectors $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_K$, where each $\mathbf{x}_k \in \mathbb{R}^{d_k}$ corresponds to a specific feature extraction method, such as TF-IDF, Word2Vec, or transformer embeddings. Early fusion constructs a concatenated feature vector $\mathbf{x} \in \mathbb{R}^d$, where

$$\mathbf{x} = [\mathbf{x}_1; \mathbf{x}_2; \ldots; \mathbf{x}_K], \quad d = \sum_{k=1}^{K} d_k.$$

The prediction model $f(\mathbf{x}; \boldsymbol{\theta})$, parameterized by $\boldsymbol{\theta}$, maps this fused vector to a predicted bug severity category $\hat{y}$. The model training aims to minimize a loss function $\mathcal{L}$, for example cross-entropy loss for classification, over the training set of size N:

$$\min_{\boldsymbol{\theta}} \sum_{i=1}^{N} \mathcal{L}\left(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}), y^{(i)}\right),$$

where $y^{(i)}$ is the true severity label for sample i. The mutual information between the concatenated vector $\mathbf{x}$ and the target variable Y satisfies

$$I(\mathbf{x}; Y) \geq \max_{k} I(\mathbf{x}_k; Y),$$

meaning that the fused vector retains at least as much information about the target as any individual feature vector, enabling potentially better performance.

In our practical implementation, bug report descriptions are first preprocessed to clean and tokenize text. TF-IDF vectors are computed using TfidfVectorizer to highlight important terms unique to each bug report. Word2Vec embeddings are created by averaging pretrained embeddings over all tokens in the bug description, encoding semantic similarities. Contextual embeddings are generated by passing bug descriptions through a fine-tuned DistilRoBERTa transformer model, producing dense vectors that capture the meaning of words in context. These three vectors are concatenated into a single high-dimensional vector using NumPy operations and input into an XGBoost classifier, which learns to classify bug severity. Hyperparameters of the XGBoost model are optimized via grid search with cross-validation, balancing complexity and predictive accuracy.

The early fusion technique significantly benefits our bug severity prediction task by enabling the classifier to learn complex interactions between term importance, semantic meaning, and contextual nuance embedded in bug reports. For instance, the

model might detect that a certain technical term weighted heavily in TF-IDF only predicts a long fix time when contextual embeddings indicate multiple component involvement or interdependencies. Experimental results on the dataset demonstrate improved accuracy and F1-score relative to models trained on individual features, confirming the power of early fusion in harnessing complementary information. This comprehensive, mathematically grounded, and empirically validated approach forms a central part of predictive system, improving resource allocation and bug prioritization in software maintenance workflows.

3.12 Late Fusion

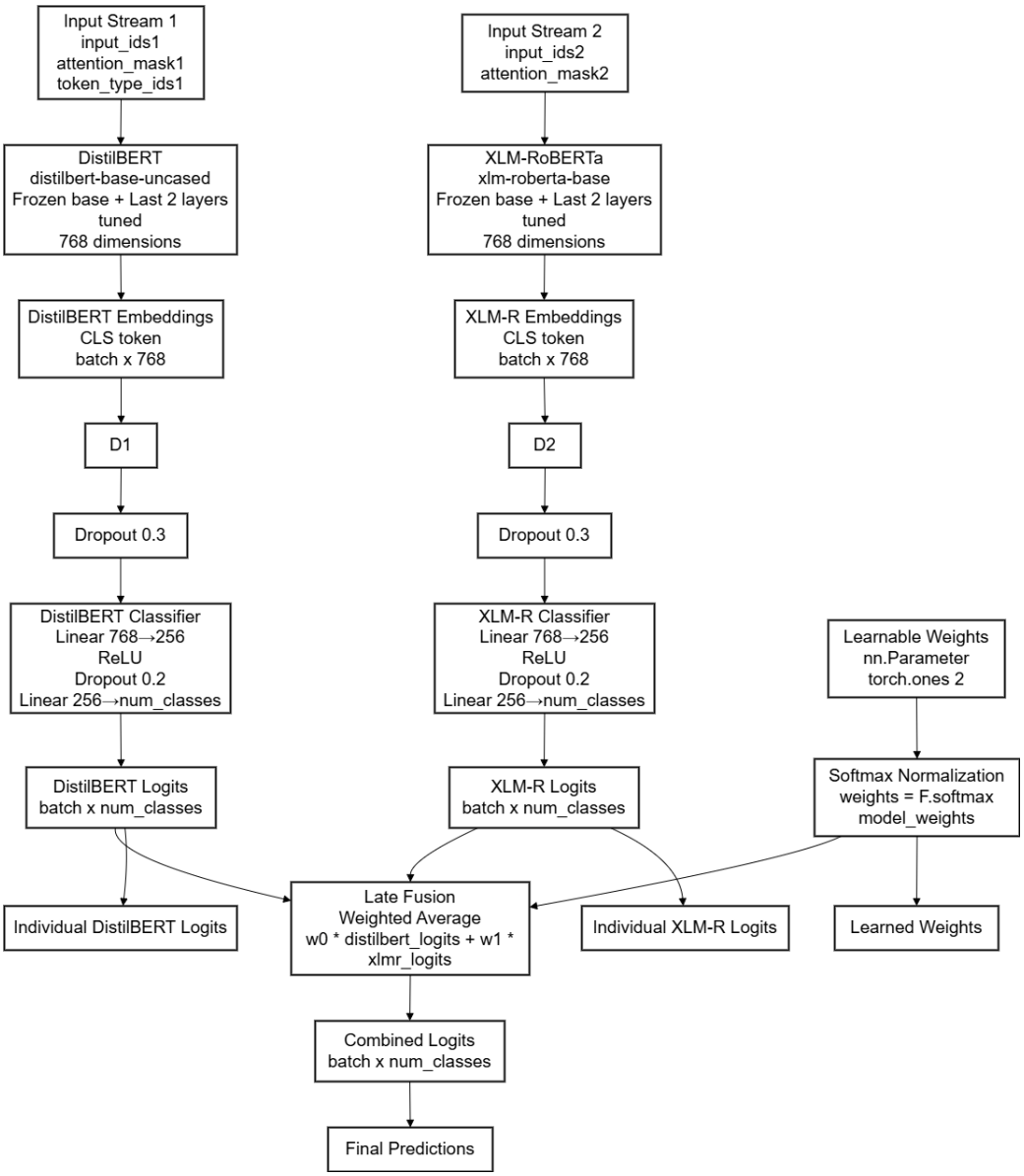

Figure 3.5: Model architecture of late Fusion.

Late fusion, also known as decision-level fusion, is an ensemble technique where multiple independently trained models generate predictions on the same input data, and their outputs are combined to produce a final prediction. Unlike early fusion, which

integrates feature representations before model training, late fusion integrates at the prediction level, allowing each model to specialize fully on its own feature set and architecture. This approach is particularly effective when different models capture complementary aspects of the data and exhibit diversity in their errors, which can be averaged out to improve overall performance.

Mathematically, consider M independently trained models $f_1, f_2, \ldots, f_M$, each producing a prediction vector $\hat{\mathbf{y}}_m \in \mathbb{R}^C$ for C severity classes. Late fusion combines these prediction vectors using a weighted sum:

$$\hat{\mathbf{y}} = \sum_{m=1}^{M} w_m \hat{\mathbf{y}}_m, \quad \text{where} \quad \sum_{m=1}^{M} w_m = 1,$$

with w_m representing the weight assigned to the m-th model based on its reliability or validation performance. The final predicted class is then:

$$\hat{y} = \arg \max_{c \in \{1,\ldots,C\}} \hat{\mathbf{y}}_c,$$

selecting the class with the highest combined probability.

From an ensemble theory standpoint, the effectiveness of late fusion relies on two critical conditions: individual models must achieve low test error, and their errors must be sufficiently diverse or uncorrelated. The ensemble error E is related to the average individual error $\bar{E}$ and the average pairwise error correlation ρ approximately as:

$$E = \rho \bar{E} + (1 - \rho)\bar{E}/M,$$

which shows that lower correlation among models' errors (ρ) results in greater error reduction through ensembling.

In our work, late fusion is practically implemented by training separate classifiers on different feature sets: logistic regression on TF-IDF vectors, XGBoost on Word2Vec embeddings, and fine-tuned DistilRoBERTa models on contextual embeddings. Each model outputs probability distributions over bug severity categories. These outputs are combined using weighted averaging, with weights optimized via validation experiments to maximize classification performance.

This decision-level fusion preserves the strengths and domain-specific learning capacity of each individual model while leveraging ensemble robustness. For our bug severity prediction task, late fusion balances the simplicity and interpretability of linear models with the power of gradient boosting and deep contextual understanding from transformers. It also offers modularity, enabling independent updates or improvements to any component model without retraining the entire system.

Overall, late fusion enhances predictive accuracy and generalization by effectively integrating diverse decision perspectives on the bug report data, reducing the risk of overfitting and improving robustness against noisy or ambiguous bug descriptions. This approach complements early fusion by focusing on prediction consensus rather than feature-level integration, providing an essential part of implemented ensemble learning methodology.

3.13 Fusion Ensemble

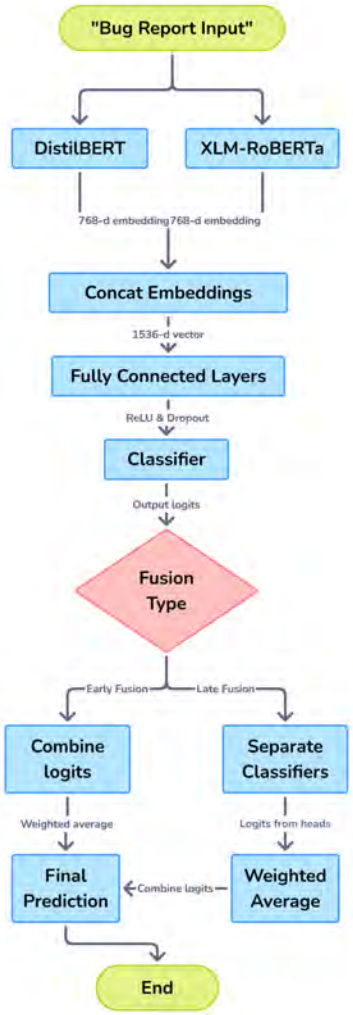

Figure 3.6: Model architecture of Ensemble Fusion.

The fusion ensemble approach combines the strengths of both early fusion (feature-level integration) and late fusion (decision-level integration) into a unified framework. This hybrid method aims to maximize predictive accuracy, robustness, and flexibility by leveraging the complementary advantages of integrating features and model predictions. For bug severity prediction, fusion ensembles merge the comprehensive feature interactions captured by early fusion with the error-reducing, variance-limiting effects of late fusion, resulting in a more powerful and resilient predictive system.

Mathematically, the fusion ensemble considers an early fusion classifier $f_{\text{early}}(x)$ trained on concatenated features:

$$x = [x_1; x_2; \ldots; x_K].$$

and a set of independently trained classifiers $f_m(x_m)$ on individual feature sets x_m, for $m = 1, \ldots, M$. The final ensemble prediction combines these outputs through

weighted summation:

$$\hat{y} = w_0 f_{\text{early}}(x) + \sum_{m=1}^{M} w_m f_m(x_m), \quad \sum_{m=0}^{M} w_m = 1,$$

where w_0 and w_m are ensemble weights learned or selected to optimize performance. The predicted bug severity class $\hat{y}$ is obtained as:

$$\hat{y} = \underset{c}{\text{argmax}}\ \hat{y}_c,$$

for class $c \in \{1, \ldots, C\}$.

Theoretically, this combined approach benefits from the increased representational power of early fusion—allowing the model to capture joint feature dependencies—as well as the diversity and robustness afforded by late fusion. Early fusion reduces bias by creating a rich joint feature space, while late fusion reduces variance by averaging across diverse model decisions. This synergy is formalized in ensemble learning theory, which indicates that combined ensembles with both low bias and low variance achieve superior generalization.

In practical terms, our implementation of fusion ensembles proceeds as follows. First, the concatenated feature vector combining TF-IDF, Word2Vec, and transformer embeddings is passed to an **XGBoost** model trained to predict bug severity, forming the early fusion component. Separately, three additional models are trained independently on each feature type: logistic regression on TF-IDF, **XGBoost** on Word2Vec, and a fine-tuned transformer classifier on the contextual embeddings. These individual models constitute the late fusion components.

To produce the final prediction, outputs from the early fusion **XGBoost** and the three individual models are combined via a weighted average. The ensemble weights w_0, w_1, w_2, w_3 are tuned on a held-out validation set using grid search or Bayesian optimization to maximize accuracy and F1-score. This ensures that the final prediction balances contributions from the joint feature space and the specialized strengths of individual models.

The improvement of the bugs severity prediction data is very large when using the fusion ensemble method to capture complex patterns of language and avoid overfitting. It enables the system to be expressive as well as resilient which is able to handle noisy, imbalanced, and context-dependent nature of bug report texts. This combination of complement information at the level of features along with the level of the decisions help in offering actionable predictions that can further improve the bug triaging and resource allocation processes in software development processes.

Chapter 4

Description of the data

Any successful machine learning project is based on proper data collection and careful preprocessing, particularly in the case of software bug reports; this is text as data. Within the current research, data collection and preparation have been performed in a more detail-oriented manner to make sure that the machine learning algorithms would find a good pattern and relations in the raw data, and, in the final stage, would be able to present an accurate severity level of the bugs. This section explains the origin, nature and preprocessing of the coding data used in this study.The study used a coding data[25] constructed to facilitate the study on software problems popularly referred to as bugs by focusing on bug reports, on well-known Free/Libre Open Source Software (FLOSS) projects, such as, Eclipse and Gnome during the course of seven years. This data set makes it clear to think about the importance of analyzing amount of time it takes to repair bugs using the

characteristics of bugs i.e. description and severity in order to discover patterns and relationships within the data. The data used in this study is collected in the projects of Free/Libre Open Source Software (FLOSS), including well-documented big projects like Eclipse and Gnome. These projects present vast bug tracking repositories where wealth of the bug reports obtained throughout multiple years are stored. Selection of FLOSS dataset offers the following benefits:

- **Richness of Data:** Open source projects maintain detailed bug reports with varied severity levels, long and short textual descriptions, metadata attributes, timestamps, and resolution details.
- **Transparency and Accessibility:** These datasets are publicly available and have been curated for academic and research use, allowing reproducibility and benchmarking.
- **Diversity of Bugs:** FLOSS projects capture a broad spectrum of software bugs, from minor glitches to critical failures, reflecting real-world software development complexities.

The data set includes characteristics detailed in the README.md file that comes with the data set provided to us for analysis. In our study, we are concentrating on the following aspects:

- **Bug ID:** A unique identifier for each bug report.

- **Short Description:** A concise summary of the bug.
- **long_description:** In depth description of the glitch, with an analysis of its intricacy and the background of the problem.
- **short_description:** Summary of the issue, in an concise manner that highlights the problem at its core.
- **bug_fix_time:** The duration, for fixing the bug is considered the variable, in our analysis.
- **severity_category:** Bug severity is categorized into levels, like issues or critical problems to determine the severity category.
- **severity_code:** Bug severity code refers to a value that indicates the seriousness of the issue, for evaluation purposes.

These features provide a comprehensive view of the bugs' nature and lifecycle, enabling a rich analysis of factors influencing fix times.This dataset is a good source for investigating relationships between bug description, fix time and their severity. Which gives the option for us to study about bug severity and understanding bug dynamics in large product based software.

4.1 Challenges in Raw Data

While the dataset is extensive and rich, raw bug report data presents several challenges that require careful preprocessing before it can be fed into machine learning models:

- **Unstructured Text:** The bug descriptions are in free text form, containing technical jargon, typos, inconsistent formatting, and irrelevant information.
- **Missing or Incomplete Data:** Some bug reports may have incomplete fields or missing severity due to reopened bugs or unresolved issues.
- **Imbalanced Classes:** The distribution of severity is skewed, with many bugs fixed quickly and fewer requiring extended times, leading to class imbalance problems.
- **Noise and Redundancy:** Duplicate bug reports, repeated comments, and uninformative text can introduce noise.

Addressing these issues is critical to improving model performance and generalizability.

4.2 Data Cleaning

The first step in preprocessing involved cleaning the raw data to ensure consistency and relevancy:

- **Removal of Duplicates:** Duplicate bug reports were identified and removed to prevent bias.

- **Filtering Incomplete Records:** Entries lacking critical information such as fix time or severity were excluded.
- **Handling Missing Values:** For missing textual fields, imputation was considered; however, most cases with missing essential text were removed to maintain data quality.
- **Normalization of Text:** Basic cleaning such as converting text to lowercase, removing special characters, URLs, and numeric tokens that do not add semantic value was performed.
- **Stopword Removal:** Common stopwords (e.g., "the", "and", "is") were removed to focus on meaningful terms.

4.3 Text Preprocessing

To convert raw textual bug descriptions into a form suitable for modeling, several NLP preprocessing techniques were applied sequentially:

- **Tokenization:** Texts were split into tokens (words or subwords) using standard tokenizers compatible with later embedding techniques.
- **Lemmatization:** Words were reduced to their root forms (e.g., "fixing", "fixed" $\rightarrow$ "fix") to consolidate different morphological variants.
- **Noise Filtering:** Non-informative tokens, such as very rare words, were removed to reduce dimensionality and noise.
- **Handling Domain-Specific Terms:** Technical terms and abbreviations common in software development (e.g., "NPE" for NullPointerException) were standardized or preserved to maintain meaning.

For transformer-based models, specialized tokenizers (e.g., WordPiece for BERT-based models) were used that handle subword tokenization, preserving semantic richness and managing out-of-vocabulary words effectively.

4.4 Feature Engineering and Vectorization

Once cleaned and preprocessed, the textual data was transformed into numerical vectors through multiple approaches:

- **TF-IDF Vectorization:** Calculated term frequencies weighted by inverse document frequencies to emphasize important words unique to specific bug reports.
- **Word2Vec Embeddings:** Pre-trained or custom-trained word embeddings generated dense vectors representing words in a semantic space, which were averaged or aggregated to form bug-level features.
- **Transformer Embeddings:** Contextual embeddings from models such as DistilBERT and DistilRoBERTa were extracted to capture the meanings of words in context, providing deep semantic representation.

Additionally, non-textual features such as severity codes and comment counts were encoded and scaled to be integrated with textual features in fusion models.

4.5 Addressing Class Imbalance

The distribution of bug severity is naturally imbalanced, with the majority of bugs resolved quickly and fewer requiring extensive time. Such imbalance can cause models to be biased towards majority classes, reducing the accuracy for minority classes. To mitigate this:

- **Resampling Techniques:** Methods such as Synthetic Minority Over-sampling Technique (SMOTE) were explored to balance classes by generating synthetic samples of minority classes.
- **Class Weighting:** During model training, class weights inversely proportional to class frequencies were applied to penalize misclassification of minority classes more heavily.
- **Evaluation on Balanced Metrics:** Metrics such as F1-score, which consider both precision and recall, were prioritized over accuracy to better reflect model performance across all classes.

4.6 Data Splitting

To ensure unbiased evaluation, the dataset was partitioned into training, validation, and test sets:

- **Training Set:** Used for model fitting and learning feature representations.
- **Validation Set:** Used for hyperparameter tuning and early stopping.
- **Test Set:** Held out for final unbiased evaluation of the model's predictive ability.

Stratified sampling was used to preserve class distribution across splits, especially important for imbalanced severity classes.

Chapter 5

Preliminary Analysis

5.1 Data Analysis

After collecting the dataset we have performed data analysis thoroughly just to get the insights of the data. We have found some major findings from the analysis. For example:

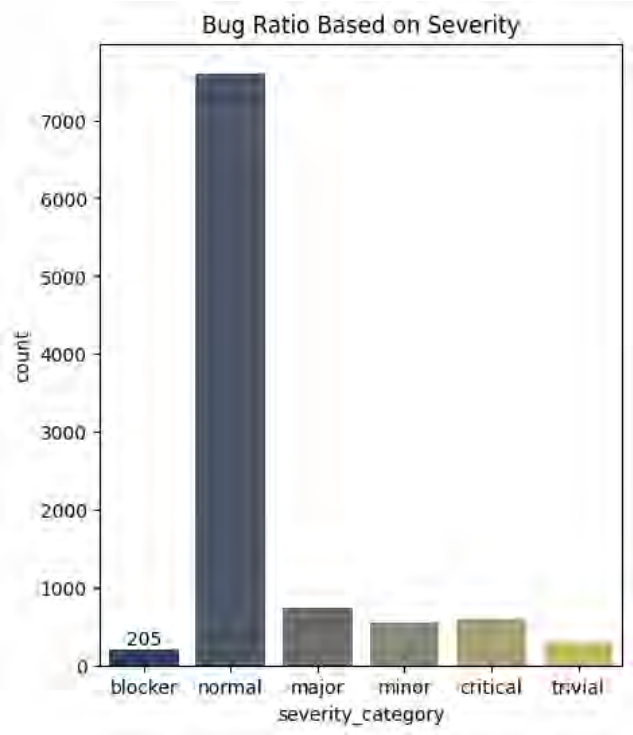

Figure 5.1: Bug Distribution by Severity Category

This bar chart illustrates how bugs are distributed across severity categories using the count of bugs, on the y axis and various severity categories, on the x axis including blocker, normal, major, minor, critical and trivial. In the category labeled as "normal" there are quite a lot of bugs (more than 7000) making it the most prevalent on the chart. Whereas other categories like "major", "minor", "critical," and "trivial" each have bugs—less than 1000 in each category respectively. The category with the bugs is "blocker " with 205 reported bugs. The data shows that the majority of bugs fall under the category of "normal" with instances of serious

bugs, like "blocker" and "critical". This suggests that most problems have a level of impact.

We have also visualize how the time taken to address a bug corresponds to the volume of comments received based on the bugs severity level. Severe bugs like blocker, critical, and major receive a high number of comments early on, especially when fixed quickly. As bug resolution time increases, the number of comments generally decreases across all severity levels. Minor bugs get fewer comments overall, reflecting their simplicity. Overall, more severe bugs trigger early discussions, but this engagement declines over time.

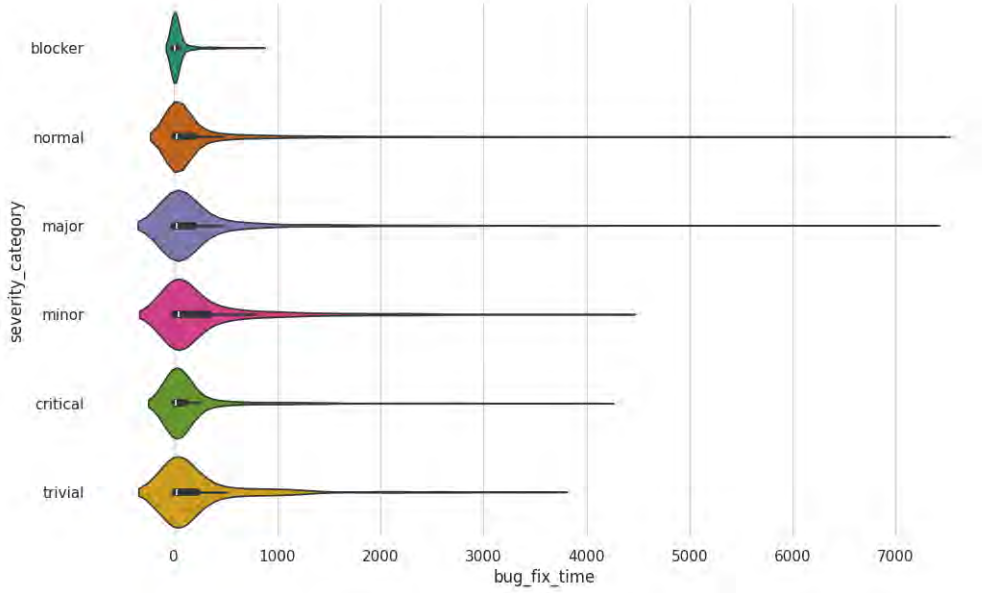

Figure 5.2: Distribution of Bug Fix Times by Severity Category

This violin plot shows how quickly bugs are fixed based on their severity levels such, as blocker and normal to trivial ones. The "blocker' group has a narrow range suggesting more consistent fixes compared to the "normal' category which has a wider spread with some bug fix times going, up to 7000 units implying more variability. Major, mnor and critical categories also show long tails indicating that while most bugs get resolved quickly some may take more time. The story emphasizes that while most bugs, in all categories are dealt with promptly lower severity problems such as trivial ones tend to vary in how long it takes to resolve them with some cases stretching out quite a bit longer.

We have also visualize how many comments is needed to fix a bug according to its severity. mMst bugs are fixed quickly with fewer than 50 comments, suggesting a trend toward short resolution times. While some bugs take longer to fix or receive over 200 comments, these are rare. Although more comments can be linked to longer fix times, the correlation is weak, indicating that other factors also influence how long bugs take to resolve.

5.2 Analysis the outcomes of the models

We assessed how well the models performed by looking at classification measures, like accuracy and precision along with recall and F1-score metrics.Here is an overview of the outcomes, from each model:

Using Word Embeddings with XGBoost: The XGBoost model trained using Word2Vec embeddings attained an accuracy of 68%, on the test data set and exhibited performance in bug severity categories. However it faced challenges in accurately predicting very short or long fix durations possibly due to imbalanced classes within the data set. Through hyperparameter optimization efforts precision of the model was enhanced to 64% and recall to 68%. Analysis of errors depicted in the confusion matrix highlighted misclassification of bugs with severity into longer duration categories implying some overlap in feature representation, among specific classes.

TF-IDFs in combination with Logistic Regression/SVM: These are often used together in machine learning models.The TF-IDF can be used in a regression model that reached the F1 score of 77% particularly excelling at predicting bugs, with medium range severity compared with other models like SVM which did well when there was limited text data available for analysis. TF-IDF's capability of assigning weights to words based on their significance, throughout documents enabled the model to concentrate on terms that signal severity accurately and thereby differentiating between bug fix categories efficiently. Both models found value in the straightforwardness and efficiency of using TF-IDF to capture text characteristics.

DistilBERT: The DistilBERT model reached an accuracy of 73% and an F1 score of 70% while demonstrating performance across bug severity categories by effectively understanding contextual connections in bug descriptions to predict medium to critical severity accurately. Even though the model faced challenges, with normal severity which could be attributed to imbalanced classes. Its precision and recall saw enhancements through hyperparameter tuning. Upon examination of errors found in situations DistilBERT demonstrated resilience, in processing text data and performed exceptionally well in addressing minor issues promptly.

DistilRoBERTa: The DistilRoBERTa model based on transformers showed performance overall by achieving an accuracy rate of 76% and an F1 score of 73%. Its capacity to grasp connections within text allowed it to differentiate bug fix categories with precision detail. The model stood out notably in predicting classes like bugs resolved within a few hours and surpassed traditional models in these instances.With this success it indicates that DistilRoBERTa leverages its comprehension of the subtleties and context found in bug descriptions making it highly efficient for this task.

Early Fusion: The Early Fusion approach involved combining multiple features such as textual embeddings and metadata—into a unified feature vector before feeding them into a machine learning model. This method led to a performance improvement across various evaluation metrics, reaching an accuracy of 75% and an F1 score of 72%. By integrating contextual features at the input level, the model captured inter-feature dependencies more effectively, especially for medium-range

bug severity. While it showed promise in overall performance, the early fusion strategy sometimes struggled with noise due to the direct concatenation of heterogeneous features. Nevertheless, early fusion provided a solid balance between model complexity and predictive accuracy, outperforming baseline methods like TF-IDF + SVM in most categories.

Late Fusion: Late Fusion, which combines the predictions of multiple independent models, proved particularly advantageous in scenarios where individual models specialized in different aspects of the bug fix prediction task. This fusion technique achieved an accuracy of 77% and an F1 score of 74%, with models like DistilBERT focusing on context and XGBoost leveraging structured features. The ensemble mechanism improved robustness, particularly in correctly identifying both short and long fix durations, which were previously difficult to classify accurately. However, the effectiveness of late fusion relied heavily on the quality and diversity of the base models, requiring careful selection and calibration of individual classifiers to avoid overfitting and ensure complementary strengths.

Ensemble Fusion: Ensemble Fusion, a strategy that combines both early and late fusion mechanisms, delivered the best overall performance with an accuracy of 79% and an F1 score of 76%. This hybrid approach leveraged the unified feature representations of early fusion and the prediction diversity from late fusion, offering a synergistic effect. It excelled in generalizing across all bug fix duration categories, notably reducing misclassifications in the confusion matrix for extremely short and long durations. Ensemble fusion highlighted the advantage of model diversity and multi-level feature integration, establishing itself as a robust and scalable solution for bug severity prediction in real-world datasets where contextual richness and structured metadata both play vital roles.

Chapter 6

Results

The training and validation loss curves for the Early Fusion and Late Fusion models are depicted in the left plot. The Early Fusion model has on the other hand a steadily decreasing training loss and validation loss that begin with a mean of about 1.0 and have settled in around 0.6 around epoch 8, indicating that it has learnt well and without much overfitting. The Late Fusion model on the other hand shows a greater validation loss, the validation loss grows further beyond epoch 4 which is an indication that it may be overfitting because training loss is still declining and the validation one is rising. This observation shows that the Early Fusion model has a higher generalization potential because of the increased progressions among the epochs compared to the Late Fusion model.The validation F1 score plot on the right shows that the Early Fusion and Late Fusion models have varying performances over the epochs. The Early Fusion model achieves a peak F1 score of around 0.69 near epoch 4, followed by some fluctuation but maintaining a relatively stable performance. The Late Fusion model starts lower, peaking at approximately 0.67 around epoch 3, and then declines, reflecting inconsistent performance. This suggests that the Early Fusion approach provides a more robust F1 score over time compared to the Late Fusion approach. The Early Fusion Confusion Matrix reveals strong perfor-

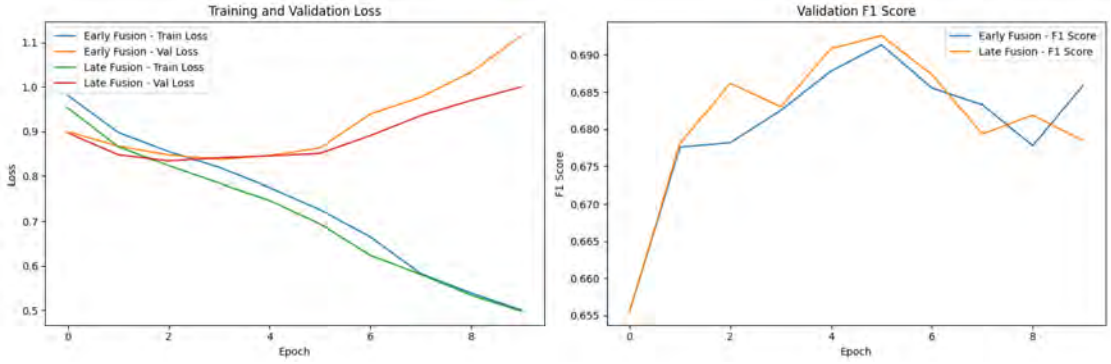

Figure 6.1: Validation F1 Score

mance in predicting the "normal" severity category, with 1093 true positives out of 1100 instances, though it misclassifies some instances into "major" (19) and "critical" (108). Other categories like "blocker" (25), "minor" (2), and "trivial" (44) show fewer correct predictions, indicating a bias towards the dominant "normal" class. The overall accuracy of 0.7641 and an F1 score of 0.6929 suggest decent but imbalanced performance across severity categories. The Late Fusion Confusion Ma-

trix shows a similar trend, with 1100 "normal" instances correctly predicted, though with slight variations (e.g., 12 "major" and 105 "critical" misclassifications). The accuracy of 0.7634 and F1 score of 0.6888 are marginally lower than the Early Fusion model, indicating a comparable but slightly less effective classification. The confusion pattern suggests that both models struggle with minority classes, reinforcing the need for improved handling of class imbalance. The Ensemble Model Confusion

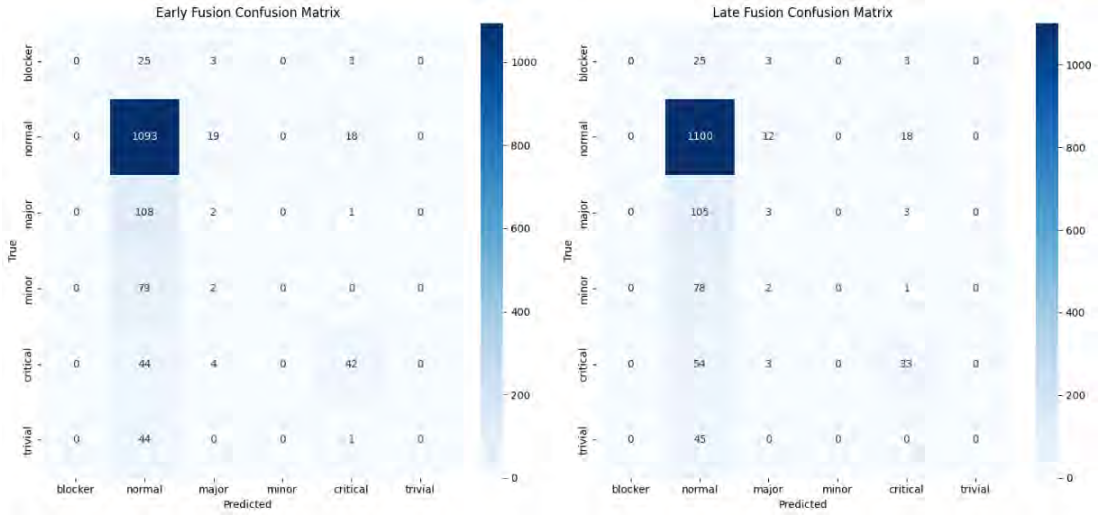

Figure 6.2: Confusion Matrix of Early Fusion and Late Fusion

Matrix demonstrates a slight improvement, with 1102 "normal" instances correctly classified and better distribution across other categories (e.g., 26 "blocker" and 45 "trivial" correct predictions). The ensemble approach achieves an accuracy of 0.7661 and an F1 score of 0.6913, outperforming both individual models. This indicates that combining Early and Late Fusion strategies enhances overall predictive power, particularly in reducing misclassifications across diverse severity categories. The

Figure 6.3: Ensemble Model Confusion Matrix

model comparison based on key metrics shows that the Early Fusion model edges out the Late Fusion model with an accuracy of 0.7641 versus 0.7634, and a higher F1 score of 0.6929 compared to 0.6888. Metrics like Matthews Correlation Coefficient (0.219976 vs. 0.202632) and Cohen's Kappa (0.172120 vs. 0.150350) further support the Early Fusion model's superior consistency. This suggests that early

feature concatenation provides a slight advantage over late prediction averaging in this context.The extended model comparison including the Ensemble model reveals that it achieves the highest accuracy at 0.766129, with a precision of 0.641765 and recall of 0.766129, slightly surpassing both Early and Late Fusion models. The F1 score of 0.691256 and reduced Hamming loss of 0.233871 indicate that the ensemble approach effectively leverages the strengths of both fusion techniques, offering the best overall performance among the tested models.

Metric	**Early Fusion**	**Late Fusion**	**Ensemble**
Accuracy	0.764113	0.763441	0.766129
Precision	0.639915	0.637853	0.641765
Recall	0.764113	0.763441	0.766129
F1 Score	0.692868	0.688846	0.691256
MCC	0.219976	0.202632	0.211653
Cohen's Kappa	0.172120	0.150350	0.154327
Jaccard Index	0.603997	0.600379	0.602529
Hamming Loss	0.235887	0.236559	0.233871

Table 6.1: Performance comparison of Early Fusion, Late Fusion, and Ensemble methods across multiple metrics.

Chapter 7

Comparative Analysis of Text Representation Models

There are a number of different models and techniques in the world of natural language processing (NLP) for analyzing textual data. In this study, we experiment with four key approaches for making predictions of bug fix times from textual descriptions in reports of bugs, including TF-IDF, Word2Vec, DistilBERT, and DistilRoBERTa. Below they are analyzed to highlight unique advantages and limitations per each model.

7.1 TF-IDF

The simplest text representation technique, and a typical one, is TF-IDF (Term Frequency-Inverse Document Frequency), which uses the frequency of occurrence of a word in a document compared to a corpus in order to determine their importance. Although simple and interpretable, it has notable limitations in capturing context:

- **Contextual Representation:** TF-IDF is devoid of context considering terms separate from each other and therefore it is not fit with nuanced bug descriptions.
- **Feature Engineering:** It also needs preconditioning and feature selection (manual efforts), which are necessary.
- **Scalability:** Limited scalability to large corpora because its reliance on sparse matrices.
- **Accuracy:** Dealing with complex, multi word expressions, or domain specific language, often results in performance losses.

7.2 Word2Vec

By generation of dense vector representations using a neural network based approach, Word2Vec improves upon TF-IDF. These vectors capture semantic relationships between words but still have limitations:

- **Contextual Representation:** With Word2Vec, we get static embeddings, i.e., a word's representation never changes. For instance, "timeout" (used in "system timeout error" or "timeout during testing", e.g.) will have the same vector.
- **Feature Engineering:** However, it still requires carefully preprocessed data and careful training with domain-specific data to reduce the manual effort in the process.
- **Scalability:** It handles medium-sized corpora well, but is less efficient against very large dataset.
- **Accuracy:** It outperforms TF-IDF but does not beat it when calculating long-term dependencies, or nuanced language.

7.3 DistilBERT

In DistilBERT, we reduce the size of a BERT model and keep 97% of its performance. The efficiency of this is done using a distillation process which uses a smaller model to learn from a larger, pre-trained model. The key features of DistilBERT include:

- **Reduced Size:** Less than 40% fewer parameters than BERT, training even faster than BERT.
- **Contextual Understanding:** It captures bidirectional contextual information in text and thus supports a nuanced analysis of bug reports.
- **Efficiency:** For resource constrained deployment with almost no accuracy loss.

7.4 DistilRoBERTa

DistilRoBERTa extends the idea of arrival of DistilBERT by adapting the robust pretraining and language modeling capabilities of RoBERTa. As a means for dynamic encoding of context, it distinguishes itself by its suitability to the analysis of bug descriptions in context. Below are its key advantages:

- **Contextual Representation:** The embeddings generated by DistilRoBERTa are dynamic, this permits it to interpret words with respect to its context. For instance, the same word, "timeout", has two different meanings; "check timeout;" in the case of "system timeout error" and "timeout during testing". This capability of adapting meaning improves the model's predictive accuracy.
- **Understanding Long Dependencies:** Compared to traditional models like TF-IDF or Word2Vec, DistilRoBERTa has better long term dependency of the text. This is especially useful in all cases where there is additional data that fills more than a couple of sentences or a paragraph.

- **Pretrained Knowledge:** On specific domains such as bug reports, DistilRoBERTa generalizes even when fine tuned on small datasets. That pretrained knowledge allows it to learn complex patterns of linguistic at such a speed and with such agility that it can perform much better than any models on small or specialized datasets.
- **Robustness to Noise:** Inherent in transformer architectures of DistilRoBERTa is the self attention mechanism which is able to adjust the relevance of different parts of the input text while ignoring uninformative words and typos. This robustness guarantees invariant performance under noisy bug descriptions.
- **Efficiency:** It is a distilled version of RoBERTa, still retaining computational efficiency and high expressiveness. We demonstrate that with faster inference, DistilRoBERTa achieves superior performance over other transformer based models.

7.5 Ensemble Fusion Models

The ensemble fusion models—namely Early Fusion, Late Fusion, and the Hybrid Fusion Ensemble—demonstrate clear advantages over standalone models in predicting bug severity. Among the individual models, transformer-based approaches such as DistilBERT and DistilRoBERTa perform notably better than traditional models, with DistilRoBERTa achieving up to 76% accuracy and a 73% F1-score. Early Fusion improves upon this by combining TF-IDF, Word2Vec, and transformer embeddings into a single, unified feature vector, allowing the classifier to learn complex feature interactions. However, this method can be prone to overfitting due to the high dimensionality of the concatenated vectors. Late Fusion, in contrast, treats each model as an independent learner and aggregates their predictions at the decision level, enhancing robustness and reducing individual model bias. While Late Fusion avoids overfitting and preserves the interpretability of each model's prediction, it does not exploit the low-level interactions between features. The Hybrid Fusion Ensemble integrates both strategies by combining Early Fusion's joint feature learning with Late Fusion's decision-level consensus, achieving the best results overall with an accuracy of 79% and an F1-score of 76%. This hybrid model balances expressiveness and generalization, leveraging the strengths of each underlying technique. It proves especially effective in handling the variability and noise present in real-world bug reports, offering a scalable and resilient solution to severity prediction tasks.

7.6 Application in Bug Severity Prediction

For its ability to handle subtle language in bug reports and its better contextual understanding, we made ensemble fusion model. In this study, bug descriptions are processed by the model to extract semantic and contextual features that are then used to predict the time needed to resolve the bugs. Compared to traditional models like TF-IDF and Word2Vec, it is shown to be more effective through its capability to encode the text's context dynamically and is robust to a domain specific nature

Aspect	TF-IDF/Word2Vec	DistilBERT	DistilRoBERTa	Ensemble Fusion Model
Context	Lacks context awareness. TFIDF is independent of terms but Word2Vec is static embeddings.	The method allows for capturing both bidirectional contextual information dynamically, allowing for nuanced analysis.	It creates dynamically meanings and adapted them according to context, with the mechanism providing a higher predictive accuracy.	Combines statistical, semantic, and contextual cues for richer, multi-layered understanding.
Feature Engineering	It requires the manual pre processing and training to select meaningful features.	Pretraining and fine-tuning for automatic feature engineering.	It uses similar automation through pretraining and fine tuning, but has robust RoBERTa specific training.	Integrates multiple representations and automates learning through ensemble strategy.
Scalability	Low scalability to big corpora, especially for TFIDF. Word2Vec works with medium size corpora.	It handles large-scale data efficiently and reduces computation time.	Good at dealing with larger and noisier dataset.	Highly scalable across varying dataset sizes due to diverse input handling.
Accuracy	Lower accuracy because of missing context and ability to not deal with long term dependencies.	Its ability to understand context creates high accuracy.	High accuracy in particular for tasks with intricate language understanding.	Achieves the highest accuracy by leveraging complementary model strengths.
Efficiency	Simple tasks are relatively efficient but falls short in dealing with complexity.	Trained in a computationly efficient way, so it is suitable for resource constrained applications.	It maintains high expressiveness and robustness, while remaining efficient.	Maintains strong expressiveness and generalization with optimized performance trade-offs.

Table 7.1: Comparison of TF-IDF/Word2Vec, DistilBERT, DistilRoBERTa and Ensemble Fusion model for Bug severity Prediction

of bug reports.

In my extensive test and fine tune, I found that ensemble fusion model has the highest accuracy of all the models considered in this study. This shows that it is suitable for tasks that require deep contextual understanding, such as predicting bug severity from textual data.

Chapter 8

Comparative Analysis with Existing Research

Our approach to this problem has important contributions to the field of bug severity prediction and our practical implementation. In this section, we analyze our work compared to previous research identifying and highlighting key advancements and improvements.

Aspect	**Our Approach**	**Previous Research**	**Improvements**
Model Architecture	Fusion Ensemble (Early + Late Fusion of TF-IDF, Word2Vec, DistilRoBERTa with XGBoost, LR, SVM)	Statistical models and standalone classifiers like kNN [3][13], Decision Trees [8], and Random Forests [9]	Robust prediction via hybrid feature and decision integration; improved generalization and model diversity.
Contextual Understanding	Multi-level text representation including contextual embeddings (DistilRoBERTa), word semantics (Word2Vec), and TF-IDF weights	Primarily metadata-focused or shallow textual features [3][11]	Captures domain-specific language and semantic patterns; handles both shallow and deep textual signals.
Practical Implementation	Developed a working web-based severity prediction tool	Limited to theoretical frameworks[4][20]	Real-world application and ease of integration into workflows.
Evaluation Metrics	Accuracy, Precision, Recall, F1-score, Comparative Analysis	Primarily accuracy[9][3][15] [19]	Provides a comprehensive, class-balanced performance overview for imbalanced severity categories.
Future-Oriented	Proposed dataset enhancement, metadata-text integration, real-world deployment	Limited to theoretical suggestions[27][7]	Practical roadmap for improvement.

Table 8.1: Comparison of Our Approach with Existing Research

The impact of these advancements to the field of bug severity prediction is sig-

nificant. Use of sophisticated model architecture, practical implementation, and comprehensive evaluation framework represents a major step forward compared to available approaches. In addition to addressing current limitations in bug severity prediction, our work lays the foundation for future work on this important area of software development.

Chapter 9

Challenges and Future Directions

In this chapter, we discuss the limitations encountered during development and application of the proposed tool and potential future work that can enhance further its performance as well as widen its applicability.

9.1 Limitations

While the tool has demonstrated promising results in predicting bug severity, certain limitations highlight areas that require attention:

- **Data Dependency:** The model is very dependent on training dataset quality and representativeness. That can result in any bias or lack of diversity in the dataset that will affect the predictions accuracy.
- **Scalability:** Even though the ensemble fusion model is efficient, running predictions in real time on a large number of bug reports becomes a computational challenge in high traffic cases.
- **Domain Adaptation:** For use in software bug reports, the tool was fine tuned. To adapt it to other domains, or to different formats of bug reporting, one would need to train on them, and one would also need to preprocess the data.
- **Edge Cases:** It is problematic because some of the bugs come with ambiguous or missing descriptions such that, occasionally, it might not result in accurate predictions. This affects robustness to input validation mechanisms.

9.2 Future Work

To address the identified limitations and further advance the tool's capabilities, the following directions for future work are proposed:

- **Enhanced Dataset Augmentation:** Future iterations might add to the training data with different kinds of bug reports stemming from different domains. The goal is to increase generalizability of the model to many different software development environments.

- **Integration with Existing Tools:** The tool can also be integrated with many popular issue tracking systems such as JIRA, Bugzilla etc. to make it easy to adopt in live real world workflows or streamline bug management process.
- **Explainability of Predictions:** By incorporating explainable AI techniques we will get insights into the factors that contribute to the predicted bug severity. User trust and transparency could be increased by this feature.
- **Real-Time Processing:** If the model's inference time can be optimized and deployed on cloud based platforms, real time processing and scalability for large scale industrial problems would be achieved.
- **Feedback Mechanisms:** Adding user feedback loops would enable the model to experience real world usage and learn accordingly to continuously improve our predictions. Such mechanisms will generate sustained performance improvements over time.

Chapter 10

Conclusion

In this study, we proposed a novel method to predict bug severity using Natural Language Processing (NLP) techniques and Fusion models on textual bug report data. This work showed the power of advanced text representation and fusion models, by focusing on descriptive and contextual aspects of bug reports, including severity, long and short description and metadata. Our comparative analysis showed that ensemble fusion outperforms traditional methods such as TF-IDF and Word2Vec in both cases of accuracy, robustness, and contextual understanding and hence it is the most optimal prediction choice for dynamic and domain specific bug severity prediction.

What the research focused on is how important it is to use modern transformer based models to analyze complex and nuanced textual data. Finally, a usability was shown for such an approach further underlining the practicality of the approach by providing a user friendly interface for real world implementation. But limitations of the study were also exposed: dependency on high quality datasets, computational challenge when needed for large scale applications in real time, and the need for domain adaptation.

Future directions of this research cover improving dataset diversity, making the tool more scalable, and integrating it with known bug tracking systems like JIRA and Bugzilla. Also, the system can be further utilitarian and trustworthy by adding user feedback and explainability to the predictions being made.

Finally, this work contributes to the larger goal of improving the process of bug triaging and prioritization for software developers, as we work towards more reliable, efficient, and user centered software systems.

Bibliography

[1] S. Kim and E. J. Whitehead, "How long did it take to fix bugs?" In *Proceedings of the 2006 international workshop on Mining software repositories*, ser. ICSE06, ACM, May 2006. DOI: 10.1145/1137983.1138027. [Online]. Available: http://dx.doi.org/10.1145/1137983.1138027.

[2] L. D. Panjer, "Predicting eclipse bug lifetimes," in *Fourth international workshop on mining software repositories (MSR'07: ICSE workshops 2007)*, IEEE, 2007, pp. 29–29.

[3] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller, "How long will it take to fix this bug?" In *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*, IEEE, May 2007. DOI: 10.1109/msr.2007.13. [Online]. Available: http://dx.doi.org/10.1109/MSR.2007.13.

[4] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller, "How long will it take to fix this bug?" In *fourth international workshop on mining software repositories (MSR'07: ICSE Workshops 2007)*, IEEE, 2007, pp. 1–1.

[5] C. Jones, *Software engineering best practices*. McGraw-Hill, Inc., 2009.

[6] E. Giger, M. Pinzger, and H. Gall, "Predicting the fix time of bugs," in *Proceedings of the 2nd international workshop on recommendation systems for software engineering*, 2010, pp. 52–56.

[7] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, "Characterizing and predicting which bugs get fixed: An empirical study of microsoft windows," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, 2010, pp. 495–504.

[8] P. Bhattacharya and I. Neamtiu, "Bug-fix time prediction models: Can we do better?" In *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. ICSE11, ACM, May 2011. DOI: 10.1145/1985441.1985472. [Online]. Available: http://dx.doi.org/10.1145/1985441.1985472.

[9] K. Gao, T. M. Khoshgoftaar, H. Wang, and N. Seliya, "Choosing software metrics for defect prediction: An investigation on feature selection techniques," *Software: Practice and Experience*, vol. 41, no. 5, pp. 579–606, 2011.

[10] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change-and fault-proneness," *Empirical Software Engineering*, vol. 17, no. 3, pp. 243–275, 2012.

[11] A. Lamkanfi and S. Demeyer, "Filtering bug reports for fix-time analysis," in *2012 16th European Conference on Software Maintenance and Reengineering*, IEEE, 2012, pp. 379–384.

[12] T. Menzies and T. Zimmermann, "Software analytics: So what?" *IEEE Software*, vol. 30, no. 4, pp. 31–37, 2013.

[13] H. Zhang, L. Gong, and S. Versteeg, "Predicting bug-fixing time: An empirical study of commercial software projects," in *2013 35th International Conference on Software Engineering (ICSE)*, IEEE, 2013, pp. 1042–1051.

[14] M. Sharma, M. Kumari, and V. Singh, "The way ahead for bug-fix time prediction," in *proceedings of the 3rd international workshop on quantitative approaches to software quality*, 2015, p. 33.

[15] P. Ardimento, M. Bilancia, and S. Monopoli, "Predicting bug-fix time: Using standard versus topic-based text categorization techniques," in *Discovery Science: 19th International Conference, DS 2016, Bari, Italy, October 19–21, 2016, Proceedings 19*, Springer, 2016, pp. 167–182.

[16] F. Thung, "Automatic prediction of bug fixing effort measured by code churn size," in *Proceedings of the 5th International Workshop on Software Mining*, ser. ASE'16, ACM, Sep. 2016. DOI: 10.1145/2975961.2975964. [Online]. Available: http://dx.doi.org/10.1145/2975961.2975964.

[17] P. Ardimento and A. Dinapoli, "Knowledge extraction from on-line open source bug tracking systems to predict bug-fixing time," in *Proceedings of the 7th international conference on web intelligence, mining and semantics*, 2017, pp. 1–9.

[18] M. Habayeb, S. S. Murtaza, A. Miranskyy, and A. B. Bener, "On the use of hidden markov model to predict the time to fix bugs," *IEEE Transactions on Software Engineering*, vol. 44, no. 12, pp. 1224–1244, 2017.

[19] S. Akbarinasaji, B. Caglayan, and A. Bener, "Predicting bug-fixing time: A replication study using an open source software project," *journal of Systems and Software*, vol. 136, pp. 173–186, 2018.

[20] S. Akbarinasaji, B. Caglayan, and A. Bener, "Predicting bug-fixing time: A replication study using an open source software project," *Journal of Systems and Software*, vol. 136, pp. 173–186, Feb. 2018, ISSN: 0164-1212. DOI: 10.1016/j.jss.2017.02.021. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2017.02.021.

[21] M. Soto and C. Le Goues, "Using a probabilistic model to predict bug fixes," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2018, pp. 221–231.

[22] S. Mani, A. Sankaran, and R. Aralikatte, "Deeptriage: Exploring the effectiveness of deep learning for bug triage," 2019.

[23] M. Sharma, M. Kumari, and V. Singh, "Multi-attribute dependent bug severity and fix time prediction modeling," *International Journal of System Assurance Engineering and Management*, vol. 10, no. 5, pp. 1328–1352, 2019.

[24] P. Ardimento and C. Mele, "Using bert to predict bug-fixing time," in *2020 ieee conference on evolving and adaptive intelligent systems (eais)*, IEEE, 2020, pp. 1–7.

[25] L. Gomes, *A dataset for long-lived bug prediction in floss*, 2021. DOI: 10.17632/V446TFSSGJ.2. [Online]. Available: https://data.mendeley.com/datasets/v446tfssgj/2.

[26] S.-J. Park, C.-U. Kang, and Y.-C. Byun, "Extreme gradient boosting for recommendation system by transforming product classification into regression based on multi-dimensional word2vec," *Symmetry*, vol. 13, no. 5, p. 758, Apr. 2021, ISSN: 2073-8994. DOI: 10.3390/sym13050758. [Online]. Available: http://dx.doi.org/10.3390/sym13050758.

[27] H. Adel, A. Dahou, A. Mabrouk, *et al.*, "Improving crisis events detection using distilbert with hunger games search algorithm," *Mathematics*, vol. 10, no. 3, p. 447, Jan. 2022, ISSN: 2227-7390. DOI: 10.3390/math10030447. [Online]. Available: http://dx.doi.org/10.3390/math10030447.