Titre: Title:	Predictive Power of Large Language Models in Bug Severity Detection: An Explorative Study
Auteur: Author:	Andressa Stéfany Silva De Oliveira
Date:	2024
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Silva De Oliveira, A. S. (2024). Predictive Power of Large Language Models in Bug Severity Detection: An Explorative Study [Master's thesis, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/58326/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/58326/
Directeurs de recherche: Advisors:	Daniel Aloise
Programme: Program:	Génie informatique

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Predictive Power of Large Language Models in Bug Severity Detection: An Explorative Study

ANDRESSA STÉFANY SILVA DE OLIVEIRA
Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie informatique

Mai 2024

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Predictive Power of Large Language Models in Bug Severity Detection: An Explorative Study

présenté par **Andressa Stéfany SILVA DE OLIVEIRA**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

Michel DAGENAIS, président
Daniel ALOISE, membre et directeur de recherche
Quentin CAPPART, membre

DEDICATION

Dedicated to my parents,
Aparecida and Ailton,
my loyal companion, Pingo,
and to my partner, Robin,
who gave me unlimited
support and love.

ACKNOWLEDGEMENTS

I extend my gratitude to my advisor, Daniel Aloise, for the invitation to join the Dorsal research team at Polytechnique Montreal. Despite the challenges posed by language and cultural differences in a foreign land, I am grateful for the opportunities to expand my horizons into the realm of Machine Learning, particularly Generative AI, inspired by the insights shared by Irving Muller.

I am also indebted to my fellow laboratory colleagues whose support have enriched both my research endeavors and personal experiences. Their collaboration and companionship have fostered a vibrant and nurturing work environment, filled not only with scientific discussions but also with moments of relaxation and friendship.

Additionally, I am also appreciative of the friendships I have cultivated in Montreal, as their encouragement and companionship have provided solace and strength throughout this academic journey.

Special thanks to my boyfriend, Robin, who worked as a research associate on my project and was instrumental in helping with the intricacies of Compute Canada. His support, motivation, and thoughtful gestures, such as invited me out for walks to alleviate stress, have been indispensable in enabling me to focus on my research with a light mind.

Finally, my heartfelt appreciation goes to my parents, Aparecida and Ailton, whose support and encouragement have been the cornerstone of my educational pursuits. From the outset, they instilled in me the values of perseverance and determination, giving me of their support and help whenever needed. I also extend my gratitude to my loyal companion, Pingo, whose comforting presence and support, even from afar, provided solace and companionship throughout this journey.

RÉSUMÉ

Les systèmes de suivi des bogues sont essentiels en génie logiciel car ils facilitent la gestion structurée des défauts et des améliorations logicielles. Ces systèmes servent d'outils pour surveiller et traiter les problèmes survenant lors des phases de développement et de maintenance des applications logicielles. Les utilisateurs utilisent les systèmes de suivi des bogues pour soumettre des rapports détaillés, décrivant les problèmes rencontrés et leur impact potentiel sur l'expérience utilisateur. Une fois soumis, ces rapports passent par une phase cruciale appelée triage des bogues, où une équipe d'experts évalue systématiquement la sévérité, la priorité et l'impact global de chaque problème signalé.

Cependant, le processus d'évaluation de la sévérité des bogues présente des défis en raison de sa nature subjective. La variabilité dans l'évaluation de la sévérité est influencée par les perceptions individuelles, les expériences et les facteurs contextuels, ce qui complique davantage le processus de triage et de résolution des bogues. De plus, le volume important de rapports de bogues générés quotidiennement au sein des plates-formes de suivi des bogues souligne le besoin pressant d'automatiser les décisions de sévérité.

Bien que des recherches antérieures aient exploré l'automatisation de la prédiction de la sévérité des bogues en utilisant des techniques de traitement du langage naturel (par exemple, la tokenisation, la suppression des mots vides, le stemming et la lemmatisation) et des algorithmes d'apprentissage machine traditionnels (par exemple, Naive Bayes, k-Nearest Neighbors et Arbres de décision), des progrès récents dans l'apprentissage en profondeur, notamment les modèles basés sur des transformateurs tels que BERT, Llama et GPT, offrent le potentiel de renforcer considérablement la précision de prédiction. Ces modèles subissent un pré-entraînement approfondi sur des données textuelles, ce qui leur permet de capturer des motifs complexes et des nuances contextuelles inhérentes au langage. Contrairement aux embeddings algorithmiques conventionnels, qui peuvent avoir du mal à capturer des relations sémantiques subtiles, les grands modèles de langage excellent dans la génération d'embeddings qui reflètent adéquatement le sens sémantique et les nuances contextuelles des mots dans les phrases et les documents.

Cette étude vise à faire progresser la prédiction de la sévérité des bogues au sein des environnements de suivi des bogues en explorant l'efficacité des grands modèles de langage, en particulier Llama 2, dans la distinction entre les bogues sévères et non sévères. Notre investigation utilise deux méthodologies : l'utilisation de modèles pré-entraînés et leur ajustement fin avec notre ensemble de données. Nous analysons les données de rapports de bogues d'Eclipse et

de Mozilla, des ensembles de données importants dans la littérature, pour évaluer l'efficacité de nos approches proposées.

De plus, notre étude compare les performances de nos méthodes basées sur de grands modèles de langage avec des algorithmes de base, y compris Naive Bayes, k-Nearest Neighbors et Support Vector Classifier. Nous utilisons une recherche aléatoire d'hyperparamètres pour optimiser les modèles de base, sélectionnant le meilleur modèle en fonction de la valeur de l'aire sous la courbe. De plus, des algorithmes de recherche aléatoire sont utilisés, et les valeurs de l'aire sous la courbe guident le choix final des meilleurs hyperparamètres. De plus, nous utilisons des matrices de confusion pour générer des métriques telles que la précision, le rappel et le score F1 pour évaluer les résultats.

L'analyse des résultats souligne la performance supérieure des modèles Llama 2 entrainés sur nos ensembles de donnés, atteignant un taux de précision de 90% pour la classification des bogues sévères sur l'ensemble de données Eclipse, surpassant les algorithmes de base avec des précisions allant de 75% à 81%. De plus, Llama 2 entrainé sur nos ensembles de donnés présente des scores F1 allant de $71,67\%$ à $72,10\%$ pour la classe 0 (non sévère) et de $91,30\%$ à $91,56\%$ pour la classe 1 (sévère), soulignant encore son efficacité. En revanche, les algorithmes de base donnent des valeurs de score F1 allant de $0,99\%$ à $46,07\%$ pour la classe 0 et de $85,30\%$ à $86,06\%$ pour la classe 1.

En résumé, cette étude contribue à l'avancement de la prédiction de la sévérité des bogues, offrant des perspectives sur l'efficacité des grands modèles de langage tels que Llama 2. Alors que le développement logiciel continue d'évoluer, l'intégration de techniques avancées d'apprentissage machine promet de rationaliser les processus de triage et de résolution des bogues, améliorant ainsi la qualité du logiciel et l'expérience utilisateur.

ABSTRACT

Bug tracking systems are essential in software engineering by facilitating the structured management of software defects and enhancements. These systems serve as tools for monitoring and addressing issues that arise during the development and maintenance phases of software applications. Users leverage bug tracking systems to submit detailed reports, outlining encountered issues and their potential impact on user experience. Once submitted, these reports undergo a crucial phase called bug triage, wherein a team of experts systematically evaluates the severity, priority, and overall impact of each reported issue.

However, the process of assessing bug severity inherently presents challenges due to its subjective nature. The variability in severity assessment is influenced by individual perceptions, experiences, and contextual factors, further complicating the bug triage and resolution process. Moreover, the sheer volume of bug reports generated daily within bug tracking systems platforms highlights the pressing need for automating severity decisions.

Although previous research has explored automating bug severity prediction using Natural Language Processing (e.g. tokenization, stop words removal, stemming, and lemmatization) techniques and traditional machine learning algorithms (e.g. Naive Bayes, k-Nearest Neighbors, and Decision Trees), recent strides in deep learning, notably transformer-based models such as BERT, Llama, and GPT, offer the potential to reinforce prediction accuracy significantly. These models undergo extensive pre-training on textual data, enabling them to capture intricate patterns and contextual nuances inherent in language. Unlike conventional algorithmic embeddings, which may struggle to capture subtle semantic relationships, large language models excel at generating contextual embeddings that adequately reflect the semantic meaning and contextual nuances of words within sentences and documents.

This study aims to advance bug severity prediction within bug tracking systems by exploring the effectiveness of large language models, particularly Llama 2, in distinguishing between severe and non-severe bugs. Our investigation employs two methodologies: utilizing pre-trained models and fine-tuning them with our dataset. We analyze bug report data from Eclipse and Mozilla, prominent datasets in the literature, to assess the efficacy of our proposed approaches.

Furthermore, our study compares the performance of our large language model-based methods with baseline algorithms, including Naive Bayes, k-Nearest Neighbors, and Support Vector Classifier. We employ a random search of hyperparameters to optimize the LLM baselines, selecting the best model based on the area under the curve value. Additionally, fine-tuning

algorithms are utilized, where the area under the curve values guides the selection process. Besides, we utilize confusion matrices to generate metrics such as accuracy, precision, recall, and F1-score to evaluate the results.

Analysis of the findings underscores the superior performance of fine-tuned Llama 2 models, achieving an accuracy rate of 90% for severe bug classification on the Eclipse dataset, surpassing the baseline algorithms with accuracies ranging from 75% to 81%. Moreover, fine-tuned Llama 2 exhibits F1-scores ranging from 71.67% to 72.10% for class 0 (non-severe) and from 91.30% to 91.56% for class 1 (severe), further highlighting its efficacy. In contrast, the baseline algorithms yield F1-Score values ranging from 0.99% to 46.07% for class 0 and from 85.30% to 86.06% for class 1.

Overall, this study contributes to the advancement of bug severity prediction, offering insights into the effectiveness of large language models like Llama 2. As software development continues to evolve, integrating advanced machine learning techniques holds promise for streamlining bug triage and resolution processes, ultimately enhancing software quality and user experience.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ACRONYMS

BERT Bidirectional Encoder Representations from Transformers

BPE Byte-Pair Encoding

BTS Bug Tracking Systems

CNN Convolutional Neural Networks

JDT Java Development Tools

kNN k-Nearest Neighbors

LDA Latent Dirichlet Allocation

LLM Large Language Model

LoRA Low-Rank Approximation

LSTM Long Short-Term Memory networks

ML Machine Learning

NER Named Entity Recognition

NLP Natural Language Processing

PCA Principal Component Analysis

POS Part-of-Speech

QLora Quantized LoRA

SVC Support Vector Classifier

SVD Singular Value Decomposition

SVM Support Vector Machines

TF-IDF Term Frequency-Inverse Document Frequency

LIST OF APPENDICES

CHAPTER 1 INTRODUCTION

Bug Tracking Systems (BTS) have become indispensable tools in software engineering, facilitating the systematic management of software defects and enhancements [1]. They are like digital command centers for software developers and project teams, helping them keep track of bugs and issues arising during software application development and maintenance. Popular BTS platforms include Bugzilla[1], Jira[2], and others. Users submit bug reports through BTS platforms providing detailed descriptions of their problems and their impact on user experience.

Once submitted, bug reports undergo a critical phase known as bug triage, where a team of experts systematically assesses the severity, urgency, and impact of each reported issue. Severity focuses on the seriousness of the issue, urgency determines how quickly it needs to be addressed, and impact assesses the breadth and depth of the problem. This process ensures that critical errors, such as frequent crashes or data loss, receive immediate attention to maintain a smoother user experience and preserve the overall quality and reliability of software applications.

Bug triage also allows project teams to prioritize efforts and allocate resources effectively. During this phase, developers, testers, or project managers review incoming bug reports, categorize them based on severity, and assign them to appropriate team members for resolution. By focusing on critical issues first and addressing less severe bugs later, the team ensures that software defects are addressed efficiently, based on their severity and impact on software functionality.

Bug reports contain detailed descriptions of encountered problems, contextual information (e.g. error messages or logs, steps to reproduce, environment Details, etc), and severity assessments assigned by reporting individuals (originators). These reports serve as valuable documentation for the bug triage process, enabling teams to make informed decisions about bug resolution priorities and resource allocation.

The bug severity assessment is inherently subjective, posing challenges to the bug triage and resolution process. The subjective nature of severity assessment introduces variability influenced by individual perceptions, experiences, and contextual factors. For instance, one individual may perceive a malfunction in a tool's autocompletion feature as highly severe, while another may regard it as a minor inconvenience [2]. This variability complicates the

[1]https://www.bugzilla.org/

[2]https://www.atlassian.com/software/jira

prioritization of bug resolution efforts and allocation of resources, as severity judgments may differ among people.

Nevertheless, bug severity levels are crucial for prioritizing bug resolution efforts and allocating resources effectively within BTS environments. Common severity levels include blocker, critical, major, normal, minor, trivial, and enhancement(feature request). Each severity level signifies the impact of a reported bug on software functionality and user experience. By categorizing bugs based on severity levels, development teams can prioritize critical issues and address them promptly to maintain software reliability and user satisfaction.

Leading organizations and communities in software development, such as Eclipse and Mozilla, have established guidelines and categorizations to standardize severity assessments and streamline bug resolution processes within their respective BTS platforms. These guidelines define severity tags associated with specific descriptions and implications for development and testing workflows. By adhering to predefined severity guidelines, development teams can manage bug triage and resolution processes, ensuring prioritization of bug resolution tasks.

Currently, there is a growing body of research focusing on automating the severity prediction of bug reports using Natural Language Processing (NLP) combined with machine learning (ML) techniques. Researchers commonly leverage textual fields such as bug summary or description to extract relevant information. Techniques such as tokenization, stop words removal, stemming, and lemmatization are applied to preprocess the text, making it suitable for analysis. Following preprocessing, the textual data is transformed into numerical representations through vectorization methods such as Term Frequency-Inverse Document Frequency(TF-IDF) or word embeddings like Word2Vec or GloVe. These numerical representations capture semantic information embedded within the text, enabling machine learning algorithms to learn patterns and relationships effectively.

Several ML models have been employed for severity prediction, including Naive Bayes [3, 4, 5, 6, 7], k-Nearest Neighbors (kNN) [8, 6, 9], decision trees [10, 7], random forests [9, 11, 7], support vector machines (SVM) [4, 12], and neural networks [13, 6, 12, 14]. Each algorithm offers unique advantages and is applied based on the dataset's characteristics and the desired prediction performance.

Moreover, severity prediction models vary in their prediction objectives. Some models focus on binary prediction [3, 15], classifying bug reports into severe and non-severe categories. In contrast, others adopt a multiclass approach [4, 16, 8, 5], categorizing bugs into severity levels such as blocker, critical, major, minor, trivial, and enhancement/feature request.

Recent advancements in deep learning techniques, particularly with the advent of transformer-

based models like BERT (Bidirectional Encoder Representations from Transformers) and its variations [17], have shown promise in improving severity prediction accuracy. These models leverage large-scale pretraining on textual data to capture intricate linguistic patterns and semantics, thereby enhancing the performance of severity prediction tasks.

A significant impetus for automating bug severity decisions stems from the substantial volume of reports generated daily within BTS platforms. For instance, within the Mozilla environment, over 350 new reports necessitate triage every day in its BTS [18, 19].

By synthesizing insights from existing literature and leveraging advancements in NLP and machine learning, this research aims to contribute to the advancement of bug severity prediction within BTS environments. The subjective nature of bug severity assessment and the challenge posed by the high demand for bug reporting in triage motivate the hypothesis that leveraging advanced natural language processing techniques can increase the accuracy of bug severity prediction compared to traditional methods and accelerate the bug triage process in BTS environments. Additionally, the research aims to explore the effectiveness of fine-tuning Large Language Models (LLMs) based on bug descriptions to distinguish between severe and non-severe bugs. By addressing the challenges posed by subjective severity assessments, the research seeks to improve bug severity prediction accuracy and enhance bug triage processes within BTS environments.

1.1 Research objectives

In this study, our research objectives are designed to investigate the efficacy of leveraging LLMs for bug severity prediction within BTS environments. We aim to address the following key objectives:

- **Evaluation of Pre-trained LLMs**
  The first objective of this research is to evaluate the performance of a pre-trained LLM in predicting bug severity levels within BTS environments. We seek to assess the ability of the pre-trained LLM to accurately classify bug reports based on severity levels extracted from projects hosted on platforms like Bugzilla. This evaluation aims to determine the baseline effectiveness of LLMs in bug severity prediction tasks.

- **Investigation of Fine-tuning LLMs**
  Our second objective is to investigate the impact of fine-tuning LLMs on bug severity prediction accuracy. We will focus on fine-tuning LLMs using bug reports sourced from Bugzilla repositories associated with prominent software projects such as Eclipse and

Mozilla. Through this investigation, we aim to explore the potential enhancements in bug severity prediction capabilities achieved through fine-tuning LLMs.

- **Comparison with Traditional Machine Learning Algorithms**
  The third objective of this research is to compare the predictive performance of LLM-based approaches with traditional machine learning algorithms commonly used for bug severity prediction tasks. We will evaluate the effectiveness of LLM-based approaches against baseline models such as Naive Bayes, k-Nearest Neighbors (kNN), and Support Vector Classifier (SVC). This comparison aims to provide insights into the relative strengths and limitations of different bug severity prediction strategies.

- **Exploration of Insights and Recommendations**
  The final objective of this research is to explore potential insights derived from the performance comparison between LLM-based approaches and traditional machine learning algorithms. Based on our findings, we aim to provide recommendations for optimizing bug severity prediction strategies within BTS environments. These insights and recommendations will contribute to advancing bug severity prediction practices and enhancing software quality assurance processes.

Pursuing these research objectives, we aim to contribute to the advancement of bug severity prediction methodologies and provide valuable insights for practitioners and researchers in software engineering.

In this study, we use Llama 2[20], a comprehensive ensemble of LLM varying in scale from 7 billion to 70 billion parameters. Developed by Meta, Llama 2 has demonstrated remarkable superiority over its counterparts, including chatGPT, PaLM-Bison, Falcon-40b-instruct, Vicuna-33b-v1.3, Vicuna-13b-v1.1, and MPT-7b-chat. Notably, it is an open-source platform with robust capabilities in language understanding and generation tasks.

The architecture of Llama 2 is supported by a comprehensive training corpus drawn from a diverse array of publicly available sources, such as the C4 dataset, GitHub, and Wikipedia. Boasting a training corpus spanning 2 trillion data tokens, Llama 2 follows a training methodology akin to previous research [21, 22] and draws inspiration from the Chinchilla scaling laws [23]. Furthermore, Llama 2 integrates architectural innovations including an expanded context length and integrated query attention mechanisms, setting it apart from its predecessor, Llama 1[24].

1.2 Thesis outline

This master's thesis outlines the structure and content of the research, focusing on bug severity prediction within BTS environments. Chapter 2, *Literature Review*, explores text mining and natural language processing techniques in bug severity prediction, evaluating existing studies and identifying methodological trends. In Chapter 3, *Methodology*, the research approach and methodologies are elucidated, covering dataset selection, preprocessing methodologies, baseline establishment, and the application of Large Language Models (LLMs) for severity prediction. Chapter 4, *Results and Discussion*, presents empirical findings derived from the experiment, offering detailed analysis, and insights. Finally, Chapter 5, *Conclusion*, synthesizes key findings and contributions, reflecting on the research journey and outlining future directions. These chapters give readers a comprehensive understanding of bug severity prediction and its significance in software development practices.

CHAPTER 2 LITERATURE REVIEW

Extensive research in existing literature has focused on refining methodologies and techniques to enhance bug severity prediction. This chapter delves into fundamental natural language processing procedures for text preprocessing and vectorization. Additionally, we explain the nature of bug reports, discuss the challenges associated with bug severity prediction, and provide a comprehensive literature overview, highlighting the techniques and algorithms employed for bug severity prediction.

2.1 Text Mining

In machine learning, algorithms typically do not directly process texts in their natural language format. As a result, preprocessing techniques are essential to refine textual data, filter out irrelevant information, and transform text into numerical representations that enable pattern recognition by the machine.

Text mining encompasses a range of methods, including tokenization, stop words removal, stemming, and lemmatization, each serving to enhance the quality and relevance of textual data. These techniques play a fundamental role in software engineering, particularly in tasks like bug severity prediction, where textual data from bug reports hold valuable insights.

Tokenization breaks down texts into individual tokens, facilitating subsequent analysis and feature extraction [25]. For example, consider the sentence *The quick brown fox jumps over the lazy dog*. After tokenization, it transforms into separate tokens: *The*, *quick*, *brown*, *fox*, *jumps*, *over*, *the*, *lazy*, and *dog*.

Stop words removal is a common preprocessing step that eliminates frequently occurring but semantically insignificant words from bug reports [25]. By filtering out words like *the*, *is*, and *and*, stop words removal reduces noise. It enhances the focus on content-carrying terms, improving the accuracy of bug severity prediction models. For instance, applying stop words removal to the sentence *The quick brown fox jumps over the lazy dog* results in retaining only content-carrying terms: *quick*, *brown*, *fox*, *jumps*, *lazy*, and *dog*.

Stemming and lemmatization further normalize word forms, unifying variations into common roots and reducing feature dimensionality for improved model performance. Stemming reduces words to their root or base form by removing suffixes or prefixes. For example, *running*, *runs*, and *ran* may all stem from the root word *run*. Unlike stemming, lemmatization ensures that the reduced form belongs to the language's dictionary. For example, *am*, *are*, and *is*

would all be lemmatized to *be*. Lemmatization helps in obtaining meaningful representations of words in a text corpus.

In addition to the preprocessing techniques mentioned previously, the field of text mining encompasses a diverse array of methodologies. Preprocessing tasks are often augmented by techniques like Named Entity Recognition (NER), which identifies entities within the text, such as names of people, organizations, and locations [26]. Part-of-Speech (POS) tagging assigns grammatical categories to each word, providing insights into the syntactic structure of sentences [27]. Dependency Parsing analyzes the grammatical structure of sentences to determine the relationships between words. Syntax analysis further delves into the syntactic structure and grammar of sentences, aiding in deeper linguistic understanding.

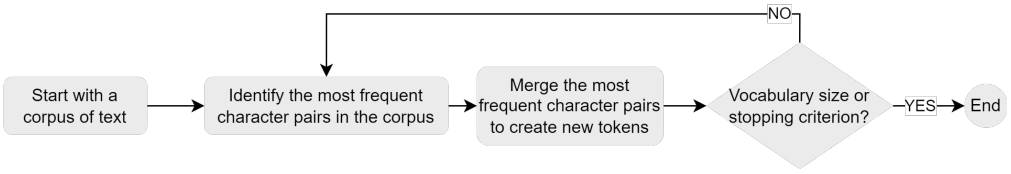
Following preprocessing, the vectorization stage transforms textual data into structured numerical representations. CountVectorizer, for instance, constructs matrices of token counts, while TF-IDF calculates the importance of terms based on their frequency and rarity across the text dataset [25]. Other vectorization techniques include Word2Vec [28], which captures semantic relationships between words as dense vectors in a continuous vector space.

Also, document embeddings aggregate word embeddings to represent entire documents as dense vectors, facilitating document-level analysis and comparison. Techniques like Doc2Vec [29] and Universal Sentence Encoder [30] are commonly used for generating document embeddings. Furthermore, topic modeling techniques, such as Latent Dirichlet Allocation (LDA), identify latent topics within a corpus, and assign documents to these topics based on word co-occurrence patterns [25]. Finally, dimensionality reduction methods like Principal Component Analysis (PCA) [31] and Singular Value Decomposition (SVD) [32] compress high-dimensional data into lower-dimensional spaces while preserving essential information.

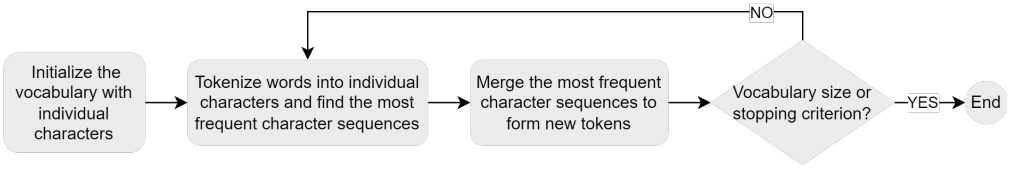
More recently, the advent of large language models (LLMs) has revolutionized the field of text mining, redefining data preprocessing methodologies, including understanding natural language. LLMs, including examples such as BERT[33], GPT[34, 35], and Llama2[20], are deep learning models trained on vast collections of text data, enabling them to learn intricate patterns and contextual nuances inherent in language. Unlike conventional algorithmic embeddings, which may struggle with capturing subtle semantic relationships, LLMs excel in generating contextual embeddings that reflect the semantic meaning and context of words within sentences and documents [36].

LLMs employ tokenization techniques that are different from traditional algorithms to process textual data. Some models, like Llama2, GPT-2, RoBERTa[37], and BART[38], use Byte-Pair Encoding (BPE)[39] for tokenization. BPE tokenization breaks down text into variable-length subword units based on their frequency in the training corpus, enabling LLMs to handle a

diverse vocabulary effectively. Similarly, models like BERT utilize WordPiece tokenization [33], a variant of BPE, which extends the concept by allowing the tokenizer to split words into smaller subword units based on their statistical significance in the textual data.



(a) BPE Tokenization Process



(b) WordPiece Tokenization Process

Figure 2.1 Subfigures depicting BPE and WordPiece Tokenization Processes

Figure 2.1a illustrates the steps involved in Byte Pair Encoding (BPE). The process begins with a corpus of text, initially represented as a set of tuples indicating word frequency. For instance [40], consider the Corpus: (1, huggingface), (1, hugging), (1, face), (1, hug), (1, hugger), (2, learning), (2, learner), (2, learners), and (1, learn), where each tuple represents the frequency of occurrence for a particular word.

In the next stage, the algorithm identifies the most frequent pairs of characters by creating an initial vocabulary composed of all unique letters in the corpus. This initial vocabulary includes elements such as *h, u, g, i, n, f, a, c, e, r, l,* and *s*. Subsequently, the algorithm counts the frequency of character pairs found in the corpus. For instance, the word *face* yields the pairs: $f+a$, $a+c$, $c+e$. This process is repeated for all words in the corpus, and the frequency of each pair is calculated accordingly. For example, the pair $f+a$ has a frequency of 2 since it appears in both *huggingface* and *face* within the corpus.

Following this step, the algorithm selects the most frequent pair, such as $l+e$, which occurs 4 times in the corpus. This pair is then merged into a single token, *le*, and added to the vocabulary. The algorithm iterates through the corpus, checking the frequency of pairs, merging, and updating the vocabulary until a stopping criterion is met. Once the vocabulary is finalized, it serves as the basis for tokenization.

In essence, the BPE algorithm enables the division of words into multiple tokens based on the contents of the vocabulary it generates. To tokenize a text using BPE, the text is segmented

into elementary units, and the merging rules are applied successively.

Figure 2.1b illustrates the WordPiece tokenizer, which shares a similar strategy with BPE but differs in the method of scoring each token candidate. To initiate the vocabulary construction process, the tokenizer segments the corpus into individual characters, placing two hash symbols (##) before characters that do not start a word. For instance [40], considering the same corpus used in the BPE example, the word *face* is segmented into *f*, *##a*, *##c*, and *##e*. After consolidating unique elementary units, the tokenizer proceeds to generate pairs present in the corpus and computes a score for each pair.

The scoring mechanism involves calculating the frequency of the pair and dividing it by the product of the frequencies of its constituent elements, see Equation 2.1. This calculation aims to penalize pairs formed by highly frequent subparts, thereby reducing their scores. For example, the pair *h+##u* would yield a score of $4/(4 \times 4) = 0.25$. This scoring process is applied to all pairs, with the highest-scoring pair subsequently merged and added to the vocabulary. The tokenizer iterates through the steps of pair creation, score calculation, merging, and vocabulary augmentation until a stopping criterion is met.

$$Score = \frac{freq\ of\ pair}{freq\ of\ first\ element \times freq\ of\ second\ element} \tag{2.1}$$

The stopping criterion signals the end of the tokenization process. If the criterion is not satisfied, the tokenizer continues iterating through the steps outlined above. Conversely, if the stopping criterion is met, the vocabulary construction phase concludes. This iterative approach enables the WordPiece tokenizer to effectively segment words into subword units based on their frequency and linguistic properties.

BPE and WordPiece tokenization are key components of LLMs' preprocessing pipelines. These tokenization strategies enable LLMs to tokenize and represent text data in a manner that captures its semantic richness and syntactic structure, laying the groundwork for advanced language understanding and processing tasks.

In summary, LLMs have elevated the standards of data preprocessing in text mining through their robust training methodologies and sophisticated tokenization techniques.

2.2 LLM Inference and Fine-tuning

In natural language processing, leveraging pre-trained LLMs for inference signifies a paradigm shift in fundamental concepts and practices. This methodology capitalizes on the extensive pre-training undergone by these models, enabling them to make predictions or execute tasks

without requiring additional training data. These models have presented remarkable performance in various tasks, including text generation, where they predict the most probable next words or phrases based on contextual cues. For instance, in a study by Lin et al. [41], LLMs were utilized to generate comments on memes, showcasing their ability to comprehend and produce coherent text. Furthermore, in language translation tasks, as demonstrated in the work by Pang et al. [42], pre-trained LLMs exhibit exceptional fluency and accuracy in translating text across different languages, underscoring their versatility in multilingual contexts.

Furthermore, pre-trained LLMs have excelled in sentiment analysis, a critical aspect of understanding the emotional tone or polarity embedded within textual data [43, 44]. By discerning sentiment, these models can classify text into positive, negative, or neutral categories, facilitating applications like social media monitoring, customer feedback analysis, and brand sentiment tracking. Additionally, in document classification tasks, pre-trained LLMs excel in categorizing textual documents into predefined classes or topics [45], thereby enhancing information retrieval and organization processes. Through their profound linguistic comprehension and context sensitivity, pre-trained LLMs have redefined the landscape of natural language processing, offering unparalleled capabilities across a spectrum of text-based applications.

Fine-tuning LLMs involves adapting pre-trained models to specific tasks or domains by further training them on task-specific data. This process allows the models to learn task-specific patterns and nuances, enhancing their performance on the targeted task. For example, in question answering tasks [46], fine-tuning LLMs involves training the model on a dataset consisting of questions and corresponding answers. During fine-tuning, the model adjusts its parameters to better understand the relationships between questions and answers, ultimately improving its ability to generate accurate responses.

Similarly, in named entity recognition tasks, fine-tuning LLMs entails training the model on annotated datasets containing named entities such as names of people, organizations, or locations. For instance, in [47], LLMs are fine-tuned to predict the synergy of drug pairs, with specific training protocols tailored to different tissue types, demonstrating the versatility and adaptability of fine-tuning techniques in diverse applications. Another example is presented in [48], where the LLM is employed to detect whether an individual's income exceeds a certain threshold based on characteristics such as age, workclass, education, relationship, and native country. To accomplish this, the LLM was fine-tuned using three credit card databases.

In addition to question answering and named entity recognition, fine-tuning LLMs has been widely employed in text summarization tasks [49]. In text summarization, the goal is to

condense large volumes of text into concise summaries while preserving essential information. Fine-tuning LLMs for text summarization involves training the model on pairs of long-form documents and their corresponding summaries. Through this process, the model learns to distill the most salient information from the input documents and generate coherent and informative summaries.

2.3 QLora

QLora [50], short for Quantized LoRA, is a technique used for efficient fine-tuning of LLMs with low hardware requirements. It combines the benefits of low-rank approximation (LoRA) [51] with quantization to enable effective adaptation of LLMs to tasks while minimizing computational resources. LoRA is a dimensionality reduction technique that approximates weight matrices in neural networks with lower-rank matrices. It achieves this by decomposing the original weight matrix into two smaller matrices, each one with reduced dimensions. This reduction in parameters helps alleviate the computational burden and memory requirements during training.

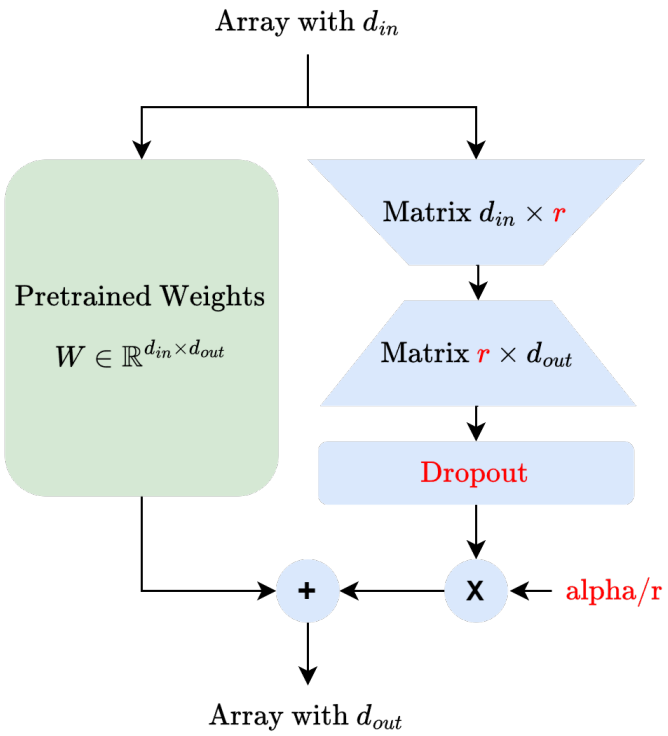


Figure 2.2 LoRA architecture

The architecture depicted in Figure 2.2 illustrates the LoRA framework. The green block

signifies the pre-trained weights of a layer, which remain frozen throughout the fine-tuning process. In contrast, the blue blocks represent the LoRA adaptation, which reduces the dimensionality of the weight matrix and add a dropout layer. The hyperparameters, namely `alpha`, `dropout`, and `r`, are highlighted in red. The `alpha` parameter governs the scaling factor, controlling the magnitude of LoRA adjustments relative to the original frozen network prediction. Meanwhile, the `dropout` parameter determines the dropout rate applied after the LoRA matrix operations, serving to regularize the model and mitigate overfitting. Lastly, the `r` parameter specifies the rank of the LoRA matrices, influencing their expressive capacity and adaptability to the underlying data patterns.

Additionally, the `r` parameter plays a significant role in the core functionality of LoRA. Assuming that a layer of the network transforms the data from d_{in} dimensions to d_{out}, LoRA reformulates this transformation as a sequence of two steps: first, from d_{in} to r, and then from r to d_{out}. This restructuring reduces the number of weights that need to be adjusted from $d_{in} \times d_{out}$ to $d_{in} \times r + r \times d_{out}$, where r represents a smaller dimensionality compared to d_{in} and d_{out}. Consequently, with a smaller value of r, the number of weights is reduced, enhancing computational efficiency while maintaining the model's expressive capacity.

After the dropout layer, the output is combined with the scaling factor $\frac{alpha}{r}$ and the output of the frozen layer. Besides, during the training process, only the LoRA adaptation matrices are updated by backpropagation, while the weight matrix of the frozen layer remains unchanged.

QLora emerges in pursuit of optimizing memory resources utilized in LLM fine-tuning. As discussed in previous paragraphs, LoRA achieves dimensionality reduction in parameter computation, contributing to computational cost reduction. However, the storage and computation data types are 16-bit BrainFloat. With QLora, the storage data type usually becomes 4-bit NormalFloat, while the computation data type remains 16-bit BrainFloat, achieving greater process optimization. Dequantization occurs for performing the forward and backward pass, but it only computes weight gradients for the LoRA parameters utilizing 16-bit BrainFloat.

Figure 2.3 illustrates the QLora architecture. The green and blue blocks represent LoRA, while the orange blocks depict the contribution of QLora. Initially, the LLM is stored in a 4-bit format. First, the weights from a layer are dequantized to the float16 format to match the input data format, enabling us to extract the output of the frozen pre-trained layer from the input data. In parallel, the LoRA adaptation process the same input through the low-rank matrices, pass the result through a dropout layer and mix this output with the one of the frozen layer using the scaling factor $\frac{alpha}{r}$. Similar to the technique presented previously, weight modifications occur only in the LoRA adaptation matrices during the forward-pass and

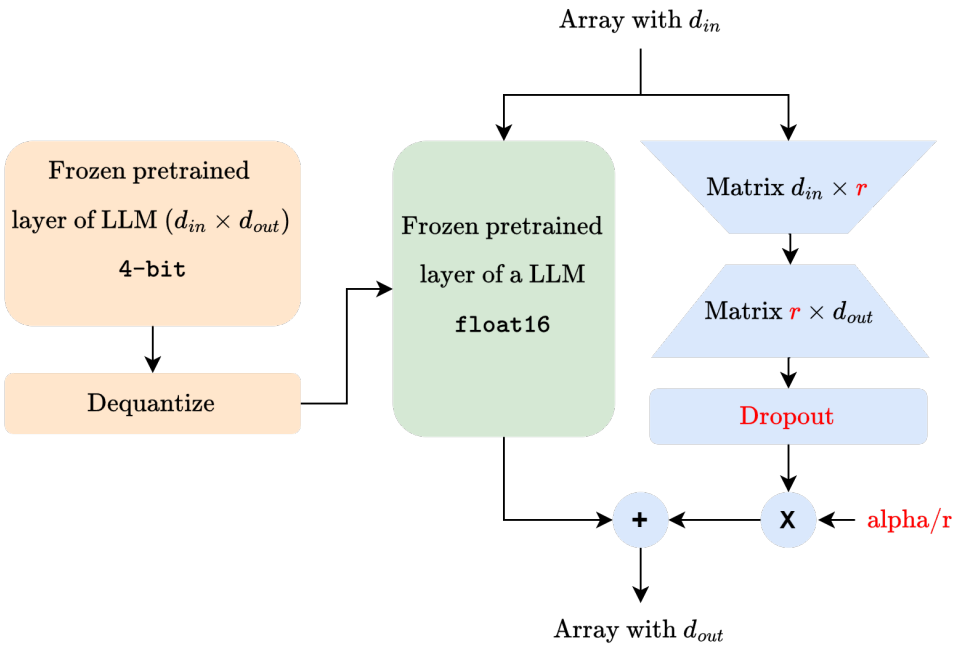


Figure 2.3 QLora architecture

backpropagation processes. Furthermore, dequantization occurs as necessary when employing a layer of the LLM network.

2.4 Bug Severity Prediction

This section provides an overview of bug reports, the challenge of bug severity prediction, and existing literature addressing this issue. Bug reports are vital documents in software development, containing detailed information about encountered issues. However, accurately predicting the severity of reported bugs remains a significant challenge. We will explore the complexities of this problem and review existing research efforts to address it.

2.4.1 Bug Report

A bug report is a structured document essential in software development and maintenance, documenting any encountered issues, anomalies, or errors within a software application [8]. Typically, it comprises predefined fields such as *Bug ID*, *Summary*, *Description*, and *Component*, among others.

The bug report depicted in Figure 2.4 is an Eclipse JDT (Java Development Tools) bug report with *Bug ID* number 9351. It was submitted by Mary Johnson and denotes its current status

as *RESOLVED* and *FIXED*. The assignment is to developer John Doe within the Eclipse Product *JDT* and Component *Core*.[1]

The report's bottom section includes information encompassing bug description, attachments (e.g., patches, test cases, or screenshots of the problem), and commentator remarks. However, commit history resides separately in the bug repository logs.

Bugs are categorized based on their *Product* and *Component* allocations. Each *Product* may encompass one or more Components. For instance, bugzilla.mozilla.org's Bugzilla Product comprises various Components like Administration, Bugzilla-General, Creating/Changing Bugs, and Documentation, among others [52].

Priority and *severity* stratify bug urgency and impact severity, respectively. The *priority* field is used to prioritize bugs, usually by the assignee or a project manager, to direct their time effectively. It ranges from P1 to P5, with P1 indicating the highest priority requiring immediate attention and P5 representing the lowest priority. It is essential to note that changing the *priority* on other people's bugs should be done with caution. For instance, in Figure 2.4, the bug has a P1 priority, indicating an urgent need for resolution.

Severity reflects the degree of impact a bug has on the system or application. Additionally, the *severity* field can also be used to indicate whether a bug is an enhancement request. The severity levels typically include blocker, critical, major, normal, minor, trivial, and enhancement. Definitions for each severity level are outlined in Table 2.1.

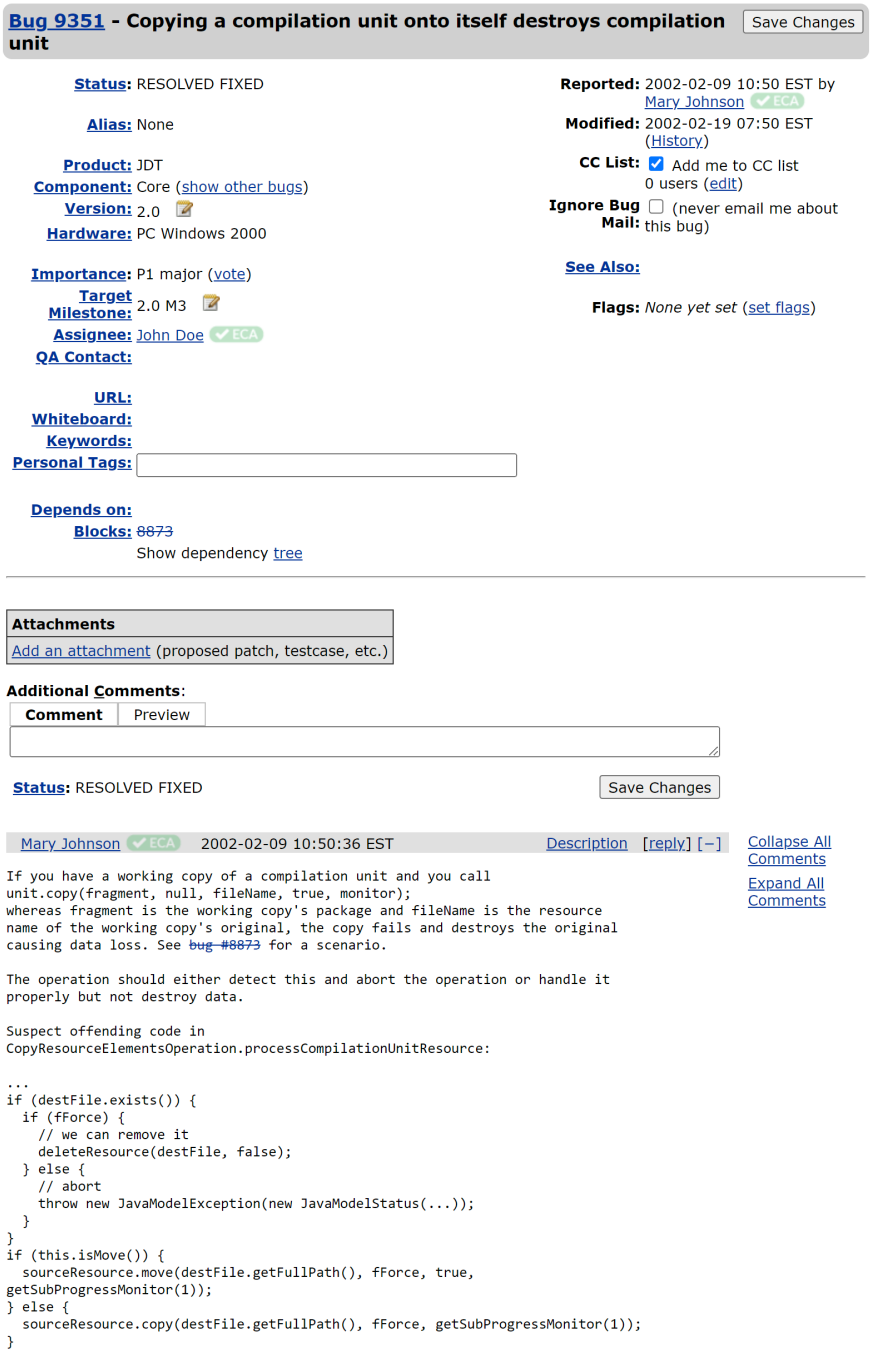
While there may be some correlation between *severity* and *priority* in certain cases (e.g., a critical bug may often be assigned a higher *priority*), they are not directly dependent on each other. For instance, a bug may have a high *severity* rating due to its significant impact on the system. Still, if it does not affect critical functionality or can be temporarily mitigated, it may be assigned a lower *priority*.

Notably, *enhancement* severity refers to new feature requests. It is excluded from the bug severity prediction in our study, as well *normal* severity, given that it is the default value of the severity field.

The Bugzilla bug lifecycle represents the sequential progression of software defects through defined stages within the Bugzilla issue tracking system [52]. This structured process ensures efficient management and resolution of reported issues within software development projects. The lifecycle typically consists of the following stages: *NEW*, *ASSIGNED*, *RESOLVED*, *VERIFIED*, and *CLOSED*.

Bugs are initially reported and enter the system with a status of *NEW*. At this stage, the

[1]Mary Johnson and John Doe are fictitious names used for illustrative purposes.



Bug 9351 - Copying a compilation unit onto itself destroys compilation unit Save Changes

Status: RESOLVED FIXED
Alias: None
Product: JDT
Component: Core (show other bugs)
Version: 2.0
Hardware: PC Windows 2000
Importance: P1 major (vote)
Target Milestone: 2.0 M3
Assignee: John Doe ECA
QA Contact:
URL:
Whiteboard:
Keywords:
Personal Tags:
Depends on:
Blocks: ~~8873~~
Show dependency tree

Reported: 2002-02-09 10:50 EST by Mary Johnson ECA
Modified: 2002-02-19 07:50 EST (History)
CC List: Add me to CC list
0 users (edit)
Ignore Bug Mail: (never email me about this bug)
See Also:
Flags: *None yet set* (set flags)

Attachments
Add an attachment (proposed patch, testcase, etc.)

Additional Comments:
Comment | Preview

Status: RESOLVED FIXED Save Changes

Mary Johnson ECA 2002-02-09 10:50:36 EST Description [reply] [–]
Collapse All Comments
Expand All Comments

```
If you have a working copy of a compilation unit and you call
unit.copy(fragment, null, fileName, true, monitor);
whereas fragment is the working copy's package and fileName is the resource
name of the working copy's original, the copy fails and destroys the original
causing data loss. See ~~bug #8873~~ for a scenario.

The operation should either detect this and abort the operation or handle it
properly but not destroy data.

Suspect offending code in
CopyResourceElementsOperation.processCompilationUnitResource:

...
if (destFile.exists()) {
  if (fForce) {
    // we can remove it
    deleteResource(destFile, false);
  } else {
    // abort
    throw new JavaModelException(new JavaModelStatus(...));
  }
}
if (this.isMove()) {
  sourceResource.move(destFile.getFullPath(), fForce, true,
getSubProgressMonitor(1));
} else {
  sourceResource.copy(destFile.getFullPath(), fForce, getSubProgressMonitor(1));
}
```

Figure 2.4 Eclipse JDT bug report (#9351)

severity field is defined, and the bug reports await triage and assignment to appropriate developers or teams for resolution. Upon assignment, bugs transition to the *ASSIGNED* status. Developers or designated personnel take ownership of the bug and begin the process of investigation and resolution.

Table 2.1 Key Concepts of Severities

Severity	Concept
`Block`	Blocks development and/or testing work. No workaround exists.
`Critical`	Crashes, loss of data, severe memory leak.
`Major`	Major loss of function.
`Normal`	Regular issue, some loss of functionality under specific circumstances.
`Minor`	Minor loss of function, or other problem where easy workaround is present.
`Trivial`	Cosmetic problem such as misspelled words or misaligned text.
`Enhancement`	Request for enhancement.

When developers believe they have addressed the reported issue, the bug status changes to *RESOLVED*. This indicates that a fix has been implemented and awaits verification. Following resolution, bugs undergo verification by testers or quality assurance personnel. If the fix is confirmed to resolve the reported issue, the bug status is updated to *VERIFIED*.

Bugs resolved and verified satisfactorily proceed to the *CLOSED* status. This signifies the completion of the bug resolution process and indicates that the issue is considered resolved and closed for further attention.

In some cases, the Quality Assurance (QA) team may not be satisfied with the solution provided, reopening the bug and changing the status *REOPEN*. In such instances, the bug is reassigned to the developer for further investigation and resolution when the status returns to *ASSIGNED*.

Throughout the lifecycle, bugs may undergo additional transitions, such as reopening if the reported issue resurfaces or requires further attention. These transitions ensure thorough tracking and management of software defects until their ultimate resolution.

In conclusion, bug reports play an indispensable role in software development and maintenance by providing developers with detailed documentation essential for bug identification, diagnosis, and resolution. While variations exist among bug tracking systems and environments, certain aspects, such as priority and severity nomenclature, remain similar across platforms. For example, the Eclipse environment (Figure 2.4) utilizes a priority scale (P1 - P5) and severity labels (trivial - blocker), though variations may occur.

Transitioning from the discussion on bug reports, our research focuses on analyzing bug report descriptions and severity fields using Natural Language Processing techniques and machine learning algorithms. By leveraging existing reports, we aim to predict the severity of bug reports, thereby enhancing software development practices and efficiency.

2.4.2 The Problem of Bug Severity Prediction

In software development, efficiently managing and prioritizing reported bugs is essential for maintaining product quality and user satisfaction. One critical aspect of bug management is predicting the severity of reported issues, which helps teams allocate resources effectively and address critical issues promptly.

Historically, bug severity assessment has been a manual process, relying on human judgment and experience to prioritize and categorize reported issues [2]. However, manual severity assignment can be subjective, inconsistent, and time-consuming. Furthermore, as software projects scale and the volume of reported bugs increases, manual assessment becomes increasingly challenging and impractical.

Automating bug severity prediction has emerged as a promising approach to address these challenges. By leveraging computational techniques and historical bug data, automated bug severity prediction systems aim to streamline bug-triaging processes and improve the efficiency of bug resolution workflows.

An often-used approach in bug severity prediction entails utilizing machine learning algorithms to analyze various features associated with reported bugs, normally focusing on bug descriptions or summaries, which encompass important information about the reported issues. These algorithms learn patterns and relationships from historical bug data to predict the severity of new bug reports.

In the literature, various approaches to bug severity prediction have been explored, ranging from traditional algorithms like Naive Bayes, k-Nearest Neighbors (kNN), and Support Vector Machines (SVM) to more advanced techniques such as Convolutional Neural Networks (CNN) and Long Short-Term Memory networks (LSTM). Initially, text mining techniques are applied, encompassing processes like tokenization, stop words removal, stemming, and lemmatization to preprocess textual data. Following this, the data undergoes vectorization, a crucial step in transforming text into numerical representations suitable for machine learning algorithms. Subsequently, bug severity prediction is performed using the selected algorithm, and evaluation metrics such as accuracy, precision, recall, and F1-score are employed to assess the model's performance and effectiveness in predicting bug severity levels.

The choice of loss function in bug severity prediction models depends on the type of prediction being performed. For binary prediction tasks, such as distinguishing between high and low severity bugs, binary cross-entropy is often used as it handles binary classification well by measuring the difference between predicted and actual outcomes. Conversely, for multiclass problems where bugs are categorized into multiple severity levels, categorical cross-entropy

is preferred. This loss function enables the model to learn the complexities of prediction across various severity levels. However, the selection of the appropriate loss function should consider factors like model architecture, data characteristics, and specific task objectives, ensuring optimal model performance and accurate bug severity predictions.

While automated bug severity prediction holds great promise, several challenges remain. These include managing data quality and volume, for instance, defaulting to a severity level of *normal* for many bug reports can skew the dataset, leading to an overabundance of reports categorized as such. This imbalance poses challenges in accurately representing the severity distribution and may hinder the model's ability to discern nuanced differences in severity levels.

Additionally, selecting and representing relevant features from bug reports is crucial for effective prediction but requires a deep understanding of contextual nuances inherent in software development. Furthermore, ensuring model interpretability is paramount, especially in critical domains where decisions based on bug severity predictions can have significant implications. Lastly, adapting to the dynamic nature of software development adds another layer of complexity, as bug reporting patterns and severity distributions may evolve over time.

Addressing these challenges requires interdisciplinary collaboration and advancements in machine learning, natural language processing, and software analytics to develop robust bug severity prediction models that enhance bug triaging, resolution, and overall software quality assurance processes.

Furthermore, as software projects evolve and new technologies emerge, the dynamics of bug reporting and severity assessment may change. Future research in bug severity prediction should consider these evolving trends and adapt predictive models accordingly.

In summary, automating bug severity prediction significantly benefits software development teams, including improved efficiency, consistency, and responsiveness in bug resolution processes. By addressing the challenges associated with manual severity assessment, automated prediction systems contribute to software products' overall quality and reliability.

2.4.3 Studies on Bug Severity Prediction

In bug severity prediction literature, machine learning methods and algorithms are widely employed. Studies typically follow a structured workflow involving data preprocessing, vectorization, and severity classification. Some focus on binary classification, distinguishing between severe and non-severe bugs, while others explore multiclass classification, categorizing severity levels like trivial, major, and blocker. Filtering mechanisms are often used to

exclude bug reports with *normal* and *enhancement* severities, enhancing prediction accuracy [3, 4, 16, 15, 5, 6, 53, 11, 9].

In [54], Menzies and Marcus undertake the task of predicting bug report severity within NASA. Their methodology begins with extracting word tokens from bug reports and applying stop word removal and stemming techniques. To identify significant tokens, they employ the TF-IDF and information gain measures.

Selected tokens are then utilized as features in a classification approach called the Ripper rule learner [55]. The Ripper rule learner is a machine learning algorithm that constructs a set of rules for classification tasks. It generates decision rules iteratively by analyzing the training data and optimizing them based on accuracy and coverage. Through this approach, Menzies and Marcus [54] assign fine-grained bug report labels corresponding to the five severity levels used within NASA's context.

Introducing a novel approach, Lamkanfi et al. [3] propose an approach for categorizing bug severities, resulting in the classification of bugs as either non-severe or severe. They define the severities *trivial* and *minor* as non-severe while categorizing reports with severity levels *major*, *critical*, and *blocker* as severe bugs. The preprocessing of bug report descriptions involves tokenization, stop word removal, and stemming techniques.

Lamkanfi et al. [3] randomly select training and evaluation sets for their experiments and employ a Naive Bayes classifier based on the probabilistic occurrence of terms (features) for bug severity prediction. Evaluation measures such as precision, recall, and AUC are utilized, with the results indicating precision and recall ranging between 0.65 and 0.85 across the three datasets used.

In the subsequent work, which builds upon the previous study [3], Lamkanfi et al. [4] compare different machine learning algorithms for bug severity classification. They demonstrate that the Naive Bayes Multinomial algorithm outperforms Naive Bayes, k-nearest neighbor, and Support Vector Machines regarding prediction performance.

Exploring a new perspective, Tian et al. [16] propose a hypothesis suggesting duplicate bug reports' use to determine the importance of different features in measuring the similarity between two bug reports. The work consists of two main components: similarity matching and assigning labels. Tian et al. leverage duplicate bug reports as training data to assign features that play a significant role in measuring report similarity. They employ a BM25 document similarity pull measure for this purpose. To predict the severity of a bug report, Tian et al. select the k closest bug reports based on the similarity measure. These k bug reports are then utilized to predict the bug report label. The severity labels used in this

study are multiclass and include blocker, critical, major, minor, and trivial. The experiments demonstrate promising results, achieving accuracies, recalls, and F-scores of up to 72%, 76%, and 74%, respectively, in predicting specific severity label classes.

In the study by Roy and Rossi [15], a range of methodologies, such as bi-gram analysis, and feature selection via the chi-squared test, are utilized to refine bug classification into severe and non-severe categories using the NB classifier. Results indicate that while incorporating bi-grams yields only marginal enhancements in classifier performance, selecting features emerges as a more influential strategy for identifying significant terms and bi-grams.

From a different perspective, Yang et al. [8] endeavors to recommend suitable developers for bug resolution tasks and forecast bug severity by extracting topics from historical bug reports and analyzing multiple features. By leveraging many features, including product, component, and priority of the bug report, the study computes the similarity between new bug reports and historical data using KL divergence. Subsequently, this similarity measure is integrated into the KNN algorithm to predict bug severity more precisely.

The study by Yang et al. [5] analyzes emotion words for bug severity prediction. The methodology involves constructing an emotion words-based dictionary to assess the emotional content of bug reports using positive and negative terms. Furthermore, the study introduces a modified machine learning algorithm derived from the Naive Bayes multinomial model to predict bug severity.

EKD-BSP [6] presents a novel approach for predicting bug report severity, leveraging keywords extracted from bug descriptions using TextRank [56]. The study advocates utilizing bug descriptions besides summary fields to enhance bug severity prediction accuracy. Comparative analysis across classifiers reveals that the Logistic Regression (LR) classifier consistently outperforms NB, KNN, and Long Short-Term Memory (LSTM) models regarding F-measure and accuracy. The FastText model [57] is employed for word representation to create two distinct embedding models: one for summaries and one for bug descriptions. Then, the LR classifier is trained using these embeddings concatenated.

In a subsequent study, Su et al. [9] present CIL-BSP, a new bug severity prediction method designed to solve class imbalance problems inherent in bug report datasets. Expanding on the foundation of EKD-BSP, CIL-BSP incorporates the techniques of Random Over-Sampling (ROS), which increases the number of instances in the minority class by randomly replicating them, Random Under-Samplings (RUS), which decreases the number of instances in the majority class by randomly removing them, and SMOTE (Synthetic Minority Over-sampling Technique) [58], which generates new instances for the minority class by interpolating between existing ones.‘Furthermore, CIL-BSP employs Logistic Regression, Random Forest[59], and

KNN classifiers to enhance its predictive capabilities further.

In [11], Dao et al. investigate the prediction performance using four tree-based algorithms: Random Forest [59], Extra Trees [60], XGBoost [61], and LightGBM [10] with Particle Swarm Optimization (PSO) to determine appropriate weights for the algorithms. LightGBM stands out in this experiment, achieving an AUC of 85.22%, recall of 83.96%, F1 score of 81.48%, and an accuracy rate of 81.28%. Additionally, XGBoost achieves the highest precision performance at 80.84%.

In their study, Hazra et al. [17] combined textual and graph-based representations for bug severity prediction. They concatenated bug descriptions and comments, generated embeddings using Doc2Vec [29], and utilized a Multilayer Perceptron (MLP) [62] for regression scores. Pre-trained sentence BERT (SBERT) models [63] contributed embeddings for bug descriptions and comments, which the MLP also processed. Two model categories were experimented with: text-based (Doc2Vec, SBERT) and graph-based (GATed models employing Sbert embeddings and graph-based models including Gat[64], GCN[65], GraphSage[66]). Results showed SBERT's superiority in higher training data scenarios (70% and 50%), while GAT outperformed SBERT significantly in lower training data scenarios ($\leq 25\%$). Evaluation metrics included Mean Absolute Error (MAE), Mean Squared Error (MSE), and Mean Absolute Percentage Error (MAPE), providing an assessment of predictive accuracy across diverse experimental setups. This comprehensive approach highlights the effectiveness of incorporating diverse representations and models, including graph-based approaches and embeddings from LLMs, for bug severity prediction tasks within software engineering activities.

In [7], source code metrics such as Lines of Code (LC), McCabe (MA), and McClure (ML) were computed and normalized using the RobustScaler algorithm. Various machine learning algorithms including KNN, SVM, Naive Bayes, Decision Tree, RandomForest, XgBoost, AdaBoost, and MLP were employed, with Decision Tree and RandomForest demonstrating superior performance. The proposed model integrates CodeBERT to process input data, generating contextual vector representations for natural language (NL) and programming language (PL) segments. These representations are combined with numerical vectors representing source code metrics and passed through a dense layer and a Softmax function for bug severity prediction. Two architectures, ConcatInline and ConcatCLS, were proposed to integrate source code metrics with CodeBERT's representations. This integration enables leveraging the strengths of both language models and source code metrics for accurate bug severity prediction.

In a new approach, Wei et al. [36] introduces a novel bug severity prediction approach called Knowledge-Intensified Contrastive Learning (KICL), which leverages pre-trained language

models and domain-specific pre-training strategies. KICL employs Knowledge-Intensified masked language modeling and contrastive learning pre-training objectives to understand project-specific bug report tokens and comprehend bug reports from a global context perspective. After conducting a comparative analysis with several pre-trained language models, including BERT [33], RoBERTa [37], and CodeBERT [67], alongside automated bug severity prediction approaches such as DNNBSP [13], BSPSO [68], PPWGCN [69], and CLBSP [14], the study demonstrated that KICL exhibited superior performance in the Precision, Recall metrics, and F1 score.

In synthesizing the studies on bug severity prediction, it becomes evident that the field encompasses diverse methodologies ranging from traditional algorithms to state-of-the-art machine learning techniques. These investigations emphasize the significance of robust preprocessing methods, meticulous feature selection, and algorithmic sophistication in accurately predicting bug severity. However, alongside these advancements, it is important to acknowledge the inherent challenges and limitations that shape the landscape of bug severity prediction, which will be discussed in the next subsection.

2.5 Challenges and Limitations of Existing Approaches

Classic machine learning algorithms have long been employed for predicting severity in bug reports, yet they exhibit inherent limitations. These algorithms often struggle to capture the intricate semantic nuances present in bug reports, leading to suboptimal prediction accuracy. Moreover, traditional approaches face challenges in handling the unstructured nature of bug report data, which includes text descriptions, user comments, and metadata. While techniques like Random Forest and Naive Bayes have shown effectiveness in certain contexts, they often fall short in comprehensively analyzing the rich textual information inherent in bug reports.

Moving beyond classical machine learning, recurrent neural networks (RNNs) have been explored for their ability to model sequential data. However, RNNs encounter practical limitations, including difficulty in encoding infinite context into finite hidden representations and challenges in parallelization, which hinder their scalability and efficiency in processing large volumes of bug reports. CNNs solve some of these challenges by enabling more efficient parallelization. However, they may still struggle with capturing long-range dependencies and semantic relationships within bug reports.

Despite CNN's advantages, the emergence of large language models (LLMs) has reshaped the landscape of natural language processing tasks. With their attention-based mechanisms

and extensive pre-training on vast corpora, LLMs possess capabilities in understanding and generating human-like text. However, adopting LLMs introduces new challenges. Issues such as model interpretability, computational resource requirements, and potential biases encoded within the training data must be carefully addressed. These challenges necessitate careful consideration and adaptation of LLMs to address bug severity prediction tasks effectively.

Shifting focus towards the practical implementation of these methodologies within BTS environments for bug severity prediction tasks, it's notable that there's a scarcity of specific studies focusing on bug severity prediction using LLMs. While some studies have successfully utilized LLMs for embeddings in bug severity prediction tasks [17, 7, 36], there's a distinct lack of research exploring the direct application of LLMs for inference and fine-tuning in this context. However, LLMs' adaptability and effectiveness in various natural language processing tasks suggest their potential efficacy in bug severity prediction. This research gap presents an opportunity for this study to contribute to the expanding body of knowledge in bug severity prediction and to advance understanding of LLMs' capabilities in real-world software development settings.

2.6 Summary and Research Directions

In this chapter, we have explored the significance of text mining and natural language processing techniques, particularly in their application to predicting the severity of bug reports. Through our exploration, we have presented bug reports' central role in software engineering, highlighting the wealth of information they contain. Predicting bug severity entails exploring the complexities of natural language processing, given the predominant use of textual data to train machine learning algorithms in existing literature.

The literature review has revealed various strategies employed for bug severity prediction, ranging from traditional machine learning algorithms to more advanced neural networks. Notably, recent investigations have begun exploring the potential of Large Language Models in this domain, particularly in generating embeddings and contextual representations.

Motivated by the current state of research and emerging trends, our study aims to contribute to this field through an exploratory work leveraging the large language model Llama2. We seek to predict the severity of bug reports, distinguishing between severe and non-severe cases. To achieve this, we will adopt two approaches: one utilizing the pre-trained model for inference and the other involving fine-tuning the model using a subset of bug reports from our dataset.

To establish a benchmark for comparison, we have selected Naive Bayes, k-NN, and SVC

algorithms as our baselines. These algorithms are widely employed in bug severity prediction tasks, offering a reliable reference point for evaluating the effectiveness of our methodology. Our evaluation will be based on the Precision, Recall, and F1-score metrics, enabling a comprehensive assessment of the performance of our approach against the established baselines.

In summary, our exploratory work aims to bridge the gap between existing methodologies and emerging trends in bug severity prediction. By leveraging advanced language models and traditional algorithms, we seek to enhance the accuracy and efficiency of bug severity prediction, contributing to the broader goals of software quality assurance and bug triaging processes.

CHAPTER 3 METHODOLOGY

In this section, we present our bug severity classification methodology, illustrating a structured progression from the input (description of bug report) to the ultimate binary prediction, where 0 signifies non-severe and 1 indicates severe. Additionally, we detail the baselines and our approaches, which consist of two different methodologies utilizing the Large Language Model - LLM. Figure 3.1 visually outlines the entire process, with blue blocks representing the baselines and green blocks representing our approach.

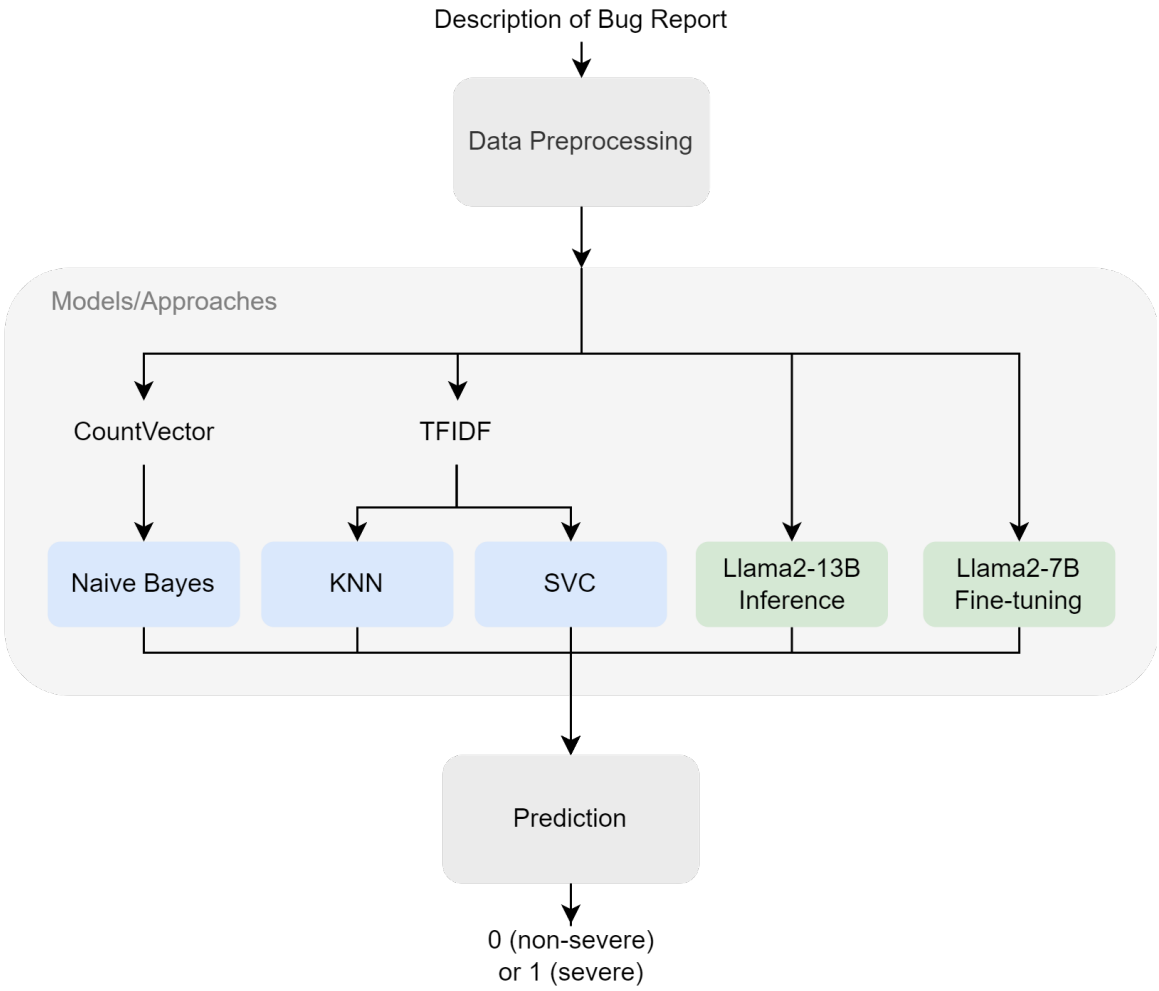


Figure 3.1 Methodology Diagram

Notice in Figure 3.1 that the general pipeline begins with *Data Preprocessing*, where raw textual information undergoes transformation to remove unnecessary text such as URLs and stopwords, facilitating more effective learning for baseline models. The subsequent paths

illustrate our baseline models: *Naive Bayes*, *kNN*, and *SVC*, optimized through a hyperparameters search technic.

The methodology extends to leverage Large Language Models (LLMs), particularly the Llama 2 model. In one approach, we utilize the pretrained HuggingFace Llama2-13B model to infer the severity of bug reports based on their descriptions. The model returns a generated text from which we extract the severity. In contrast, for a different approach, we employ the frozen pretrained HuggingFace Llama2-7B model and fine-tune it, providing severe and non-severe bug reports. In this latter approach, the classification model yields scores indicating the probabilities of each class, enabling us to determine the class based on a predefined threshold. Ultimately, both approaches yield a binary prediction: 0 for non-severe and 1 for severe outcomes.

The upcoming subsections will provide detailed insights into stages of the process. Beginning with *Data Preprocessing*, we will outline the natural language techniques employed in preprocessing bug report descriptions. Subsequent sections will focus on the creation and optimization of baseline models—Naive Bayes, kNN, and SVC—highlighting our selection of hyperparameters for each model and their significance. Then we explain the algorithm used to choose the hyperparameters values. Finally, we delineate our primary approaches involving Large Language Models through either inference and fine-tuning.

The experiments were conducted using high-performance computing resources provided by Compute Canada. Specifically, we used the sever Cedar, featuring NVIDIA V100 GPUs with 32GB of VRAM, and the server Naval, equipped with NVIDIA A100 GPUs boasting 40GB of VRAM. These GPU resources facilitated efficient training and evaluation of deep learning models.

3.1 Datasets

We leverage two widely-used open datasets, Eclipse and Mozilla collect from Bugzilla, which are extensively studied in the literature for bug severity prediction tasks [11, 53, 3, 4, 15]. These datasets contain bug reports with various attributes like Bug ID, Summary, Description, Severity, Priority, and Resolution. Initially, the Eclipse dataset comprises $361,006$ samples, while Mozilla contains $768,335$ samples. These datasets contain a diversity of bug reports, each characterized by distinct severity levels.

We used these datasets as a proof of concept to test whether our proposed framework is effective in predicting bug severity. By applying our framework to these extensive datasets, we aimed to validate their accuracy, scalability, and generalizability across different bug reports.

This approach allowed us to simulate real-world conditions and evaluate our framework's performance in various scenarios.

In the Eclipse dataset, the number of bug reports across severity levels is as follows: normal (242,819), enhancement (45,499), major (35,367), minor (13,438), critical (12,845), blocker (6,223), and trivial (4,815). Similarly, in the Mozilla dataset, we have the severity level distribution: normal (523,540), major (71,084), critical (65,478), enhancement (43,689), minor (36,617), trivial (16,134), and blocker (11,793), as shown in Figure 3.2. Notably, there is a predominance of normal severity in both datasets, often leading to its classification as noise in many studies [3, 53, 11, 16, 15, 6]. This large number of normal severities can be partially attributed to the default assignment of the normal severity value in the severity field.

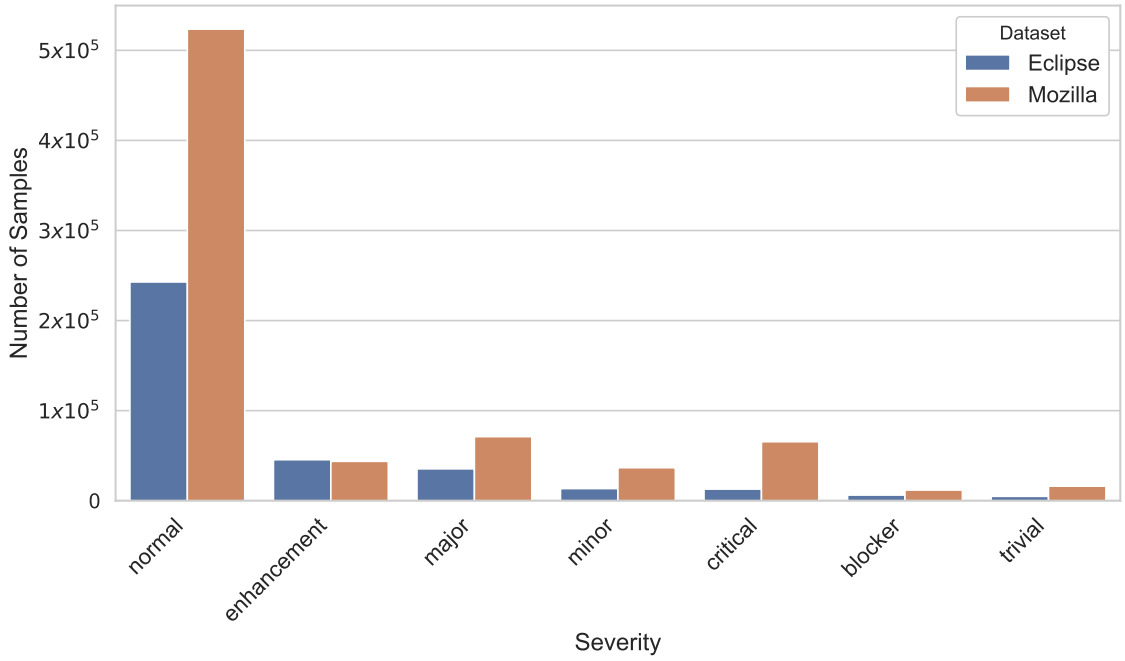


Figure 3.2 Distribution of Severities in Eclipse and Mozilla Datasets

Our datasets span bug reports from extensive time frames: the Eclipse dataset includes reports from October 10, 2001, to December 31, 2013, while the Mozilla dataset spans from September 9, 1994, to December 31, 2013.

3.2 Data Preprocessing

Before applying the baseline algorithms and the LLM, the datasets are preprocessed as illustrated in Figure 3.3. The goal of this step is to remove irrelevant descriptions, ensuring the

data is prepared for model input.

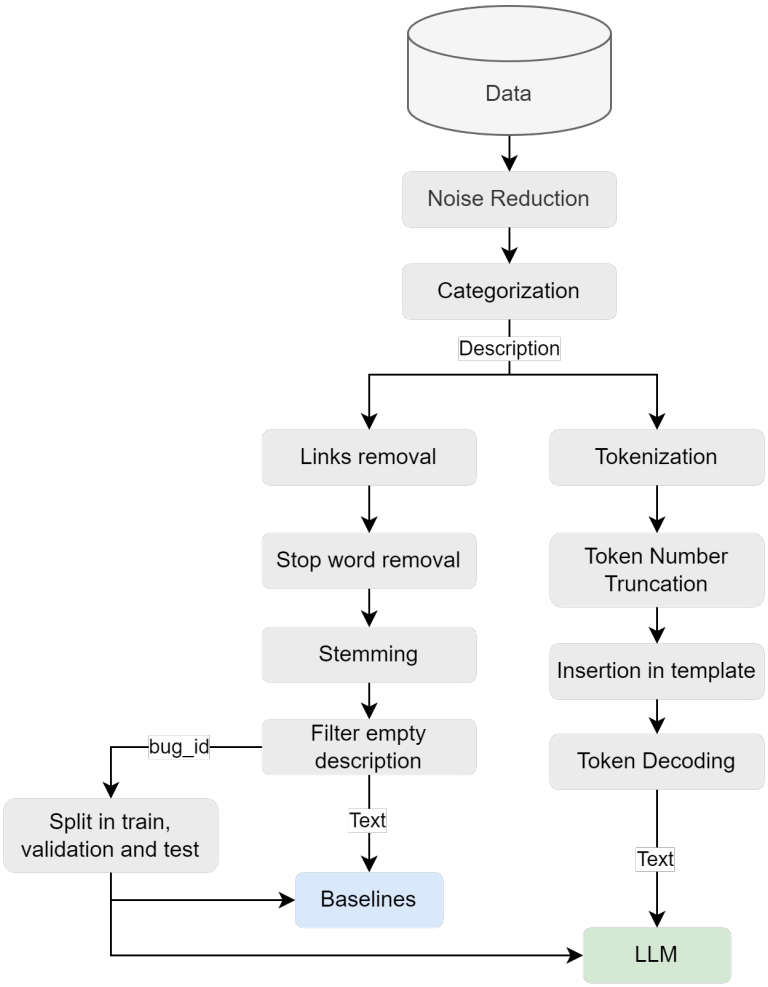


Figure 3.3 Data Preprocessing Workflow

Initially, a noise reduction strategy is applied, which removes bug reports with severity labelled as *normal* or *enhancement*. Following this, severities were categorized into two groups: non-severe and severe. The *non-severe* label, denoted by 0, encapsulates *trivial* and *minor* severities, while the *severe* label, represented by 1, encompasses *major*, *critical*, and *block* severities.

For the baseline algorithms, we first eliminate URLs (Uniform Resource Locators). The text is then subjected to two text processing techniques: stop-word removal and stemming. Stop-word removal involves eliminating common words (stop words) that do not significantly contribute to the meaning of the text. This step helps the model focus on ideas rather than specific words and removes *noise* from the text, thereby enhancing the efficiency of subsequent models. Some examples of removed words are: articles (a, an, the), conjunctions

(and, but, or), and prepositions (in, on, at, by).

Simultaneously, stemming is employed to reduce words to their root or base form. For example, the word *running* changes to *run*, helping in the normalization of variations of words and focusing on the ideas and meanings behind each word. These baseline preprocessing choices were also adopted in previous works, ensuring a methodologically sound approach [3, 4, 5, 70, 6].

After filtering out descriptions with empty content, we divide the dataset into training (70%), validation (20%), and test (10%) sets. The split is stratified to maintain consistent class proportions across subsets. For instance, if 10% of instances belong to the *non-severe* class in the entire dataset, the same ratio is preserved within each subset. This uniform distribution ensures fair comparisons of model performance. Figure 3.4 illustrates the dataset split.

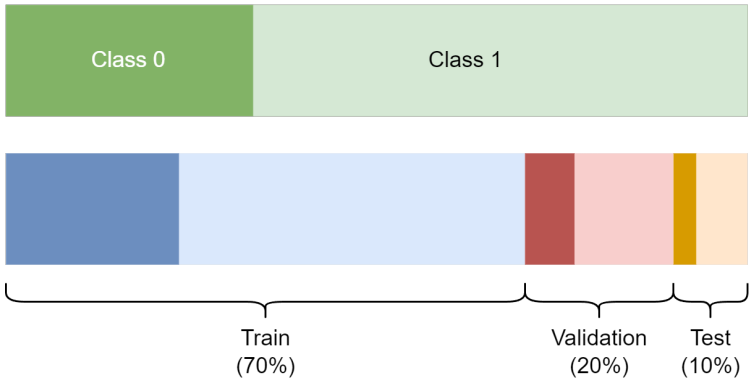


Figure 3.4 Stratified Splitting for Balanced Train, Validation, and Test Sets.

Meanwhile, the path leading to the Large Language Model (LLM) involves different text processing steps. After the initial preprocessing, the data undergoes tokenization, truncation of token numbers, insertion into a template, and subsequent token decoding. We utilize the Llama tokenizer, which uses Byte-Pair Encoding (BPE). BPE is an algorithm that compresses and represents text by merging the most frequent pairs of characters or subwords iteratively. This adaptive approach is particularly effective for handling rare words and variations in word forms within natural language, contributing to the robustness of our LLM.

Regarding the truncation process, we impose a token limit of 1000 for Llama2-13B, which is primarily used for inference with the pretrained model, and 500 for Llama2-7B, which is utilized for fine-tuning. These limits were determined through testing because of GPU memory constraints. The smaller token size for Llama2-7B reflects its use in fine-tuning, a process that heavily utilizes GPU memory. After truncation, we merge the preprocessed descriptions with the prompt template. This last step aims to provide clear instructions for

the model.

In the context of using LLMs, a prompt template serves as a structured guide or format for inputting information into these models. A prompt template is important because of its ability to provide clear and consistent instructions to the model. By following a standardized format, users can ensure that the LLM interprets their inputs accurately and responds appropriately. This structured approach also helps organize and format the input data. For our study, we utilized two prompt templates, as detailed in Appendix A.

3.3 Baseline Models

For the baseline algorithms, we selected Naive Bayes, Support Vector Classification (SVC), and K-Nearest Neighbors (kNN). These algorithms were chosen due to their common usage in the literature for bug severity classification [3, 4, 8, 6, 9].

In the case of Naive Bayes, following the initial data preprocessing, we use count vectorizer to transform the data format before providing it into the model, as shown in Figure 3.1. Count vectorizer is a technique that converts a collection of text documents into a matrix of token counts, representing the frequency of each word in the dataset. This approach allows the transformation of textual data into a format suitable for Naive Bayes.

For SVC and kNN, Term Frequency-Inverse Document Frequency (TF-IDF) is applied to vectorize the data format. TF-IDF is a numerical statistic that evaluates the importance of a word within a document relative to its occurrence across the entire dataset. Algorithms like kNN and SVC benefit from understanding the importance of terms, which is captured effectively by TF-IDF. Furthermore, kNN and SVC can be sensitive to high-dimensional spaces, and TF-IDF helps reduce dimensionality by weighting common terms.

The subsequent steps in each methodology of baselines follow a similar pattern: the reformatted data is used for training and validation, and the resulting model is then evaluated with the test data. In addition, we fine-tune each model by optimizing hyperparameters through a random sampling to improve performances. This process involves random trials based on an objective, and for that, we use Python's Optuna library. In Table 3.1, we present the algorithms along with their corresponding hyperparameter variations, which will be discussed in the subsequent subsections.

The chosen metric for optimization is the Area Under the Curve (AUC), ideal for assessing classification models as it helps identify the most accurate model. The objective of hyperparameter tuning is to maximize the AUC, ensuring optimal performance in distinguishing between positive and negative classes.

We conduct random sampling to explore hyperparameters, aiming to maximize the objective and enhance experimental robustness. Each algorithm undergoes 50 trials, aiming to optimize its performance in each attempt.

The following subsections will provide detailed information about the hyperparameters used for each model in the baseline, showing the specific configurations that underwent hyperparameter optimization by random sampling.

Table 3.1 Variation of hyperparameters optimized by random sampling

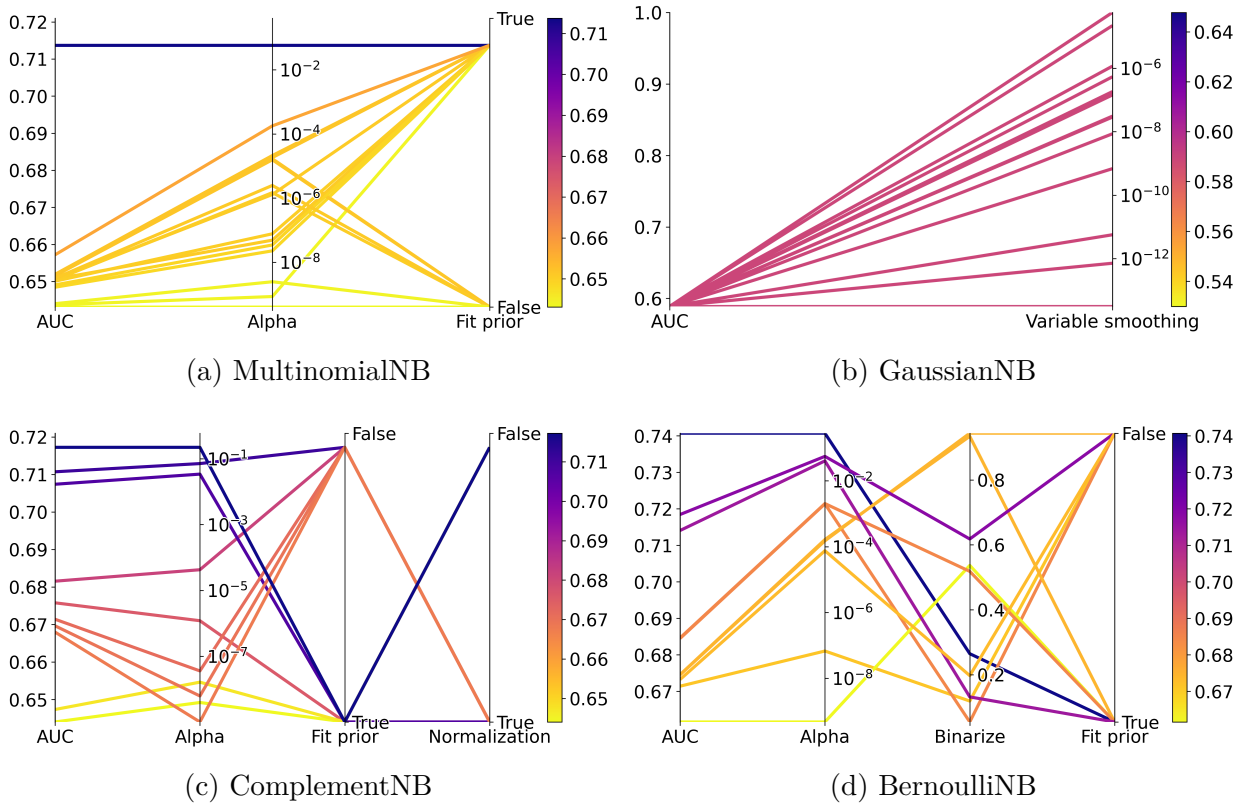
Classifier name		Hyperparameters
Naive Bayes	Multinomial	alpha: $1e-10$ to 1;fit prior: `True` or `False`.
	Gaussian	variable smoothing: $1e-14$ to 1.
	Complement	alpha: $1e-10$; fit prior: `True` or `False`; norm: `True` or `False`.
	Bernoulli	alpha: $1e-10$; binarize: 0 to 1; fit prior: `True` or `False`.
kNN		algorithm: brute, K-d tree, ball tree, or auto; leaf size: 10 to 50; n-neighbors:1, 5 or 7; p: 1, 2, or 3; weights: distance or uniform.
SVC		C: $1e-5$ to $1e5$; class weight: balanced or none; coef0: -1 to 1; decision function shape: ovr or ovo; gamma: auto or scale; kernel: sigmoid, rbf, poly or linear; max iter: 1 to 1000; shrinking: `True` or `False`; degree: 1 to 5; tol: $1e-5$ to $1e-1$.

3.3.1 Naive Bayes

The Naive Bayes family includes various algorithms, each tailored to different data characteristics and modeling assumptions. In our experimentation, we explored MultinomialNB, GaussianNB, ComplementNB, and BernoulliNB, each possessing unique traits that make them suitable for specific scenarios.

Figure 3.5 depicts a parallel coordinate plot illustrating the results of hyperparameter optimization using random sampling for Naive Bayes algorithms—MultinomialNB, GaussianNB, ComplementNB, and BernoulliNB. In this graphical representation, each line represents a unique combination of hyperparameter values explored during the optimization process.

The x-axis of the plot consists of parallel axes, each dedicated to a specific hyperparameter (e.g., *alpha*, *fit prior*, *priors*, etc.). The y-axis along each parallel axis corresponds to the values of the respective hyperparameter for a particular combination. The color gradient of the lines within each group signifies the values of the Area Under the Curve (AUC), with



(a) MultinomialNB

(b) GaussianNB

(c) ComplementNB

(d) BernoulliNB

Figure 3.5 Naive Bayes hyperparameters with 1000 samples from the Mozilla dataset

darker shades indicating higher AUC scores. The y-axis of the *classifier_name* plot includes a pink region highlighting selected trials.

This parallel coordinate plot offers a visual overview of the influence of different hyperparameter combinations on the AUC performance for each Naive Bayes algorithm. Lines that converge or exhibit similar patterns across multiple axes indicate configurations that share common values, while variations highlight distinct hyperparameter choices.

For *MultinomialNB*, default hyperparameters include *alpha*, *force alpha*, *fit prior*, and *class prior*. Figure 3.5a illustrates the variation of the hyperparameters we use in the random search for the MultinomialNB algorithm.

The hyperparameter *alpha* controls the strength of regularization through additive smoothing (Laplace/Lidstone). In our optimization, we vary *alpha*, allowing it to range from $1e-10$ to 1 within a logarithmic scale. We choose not to include *force alpha* in the optimization process because we vary *alpha* across a sufficiently broad range, rendering *force alpha* unnecessary.

Additionally, *fit prior* determines whether class priors should be adjusted based on the data.

We explore both *True* and *False* values for *fit prior*. Importantly, we choose not to include *class prior* in the optimization process. This decision is deliberate, as the use of *fit prior* renders *class prior* irrelevant. Including both could introduce unnecessary complexity without significant performance gain for our specific bug severity classification task.

Similarly, as depicted in Figure 3.5b, the *GaussianNB* algorithm employs default hyperparameters such as *priors* and *var smoothing*. In our experimentation, we focus on adjusting the *var smoothing* hyperparameter, which regulates the proportion of the largest variance of all features incorporated to ensure calculation stability. We vary *var smoothing* logarithmically from $1e-14$ to 1. It's worth noting that we opt not to vary the *priors* hyperparameter in *GaussianNB*, which signifies the prior probabilities of the classes and is typically set to *None* by default, allowing the algorithm to estimate class priors from the training data.

The *ComplementNB* algorithm is characterized by default hyperparameters, including *alpha*, *force alpha*, *fit prior, class prior*, and *norm*. Nevertheless, our focus here is on the variations of *alpha*, *fit prior*, *prior*, and *norm* during the optimization process, depicted in Figure 3.5c.

As previously mentioned, *alpha* undergoes variation within the interval of $1e-10$ to 1 on a logarithmic scale, controlling the strength of additive smoothing. For *fit prior*, we explore both $True$ and $False$ values, assessing whether *class priors* should be adjusted based on the data. The hyperparameter *prior* denotes the prior probabilities of the classes. In our experiments, we chose the probability between 0 to 1 for the first class, with the second being its complement. The hyperparameter *norm* is also adjusted, considering both $False$ and $True$ values. This adjustment determines whether feature-based normalization should be applied.

Lastly, for the *BernoulliNB* algorithm, default hyperparameters consist of *alpha*, *force alpha*, *binarize*, *fit prior*, and *class prior*. Our experimentation focuses in on *alpha*, *binarize*, and *fit prior*, revealing the dynamics of these hyperparameters, as illustrated in Figure 3.5d.

Similar to previous approaches, *alpha* and *fit prior* vary within the interval of $1e-10$ to 1 on a logarithmic scale, controlling the strength of additive smoothing and determining whether class priors should be adjusted based on the data, respectively. For the hyperparameter *binarize*, we vary between 0 and 1. The *binarize* hyperparameter influences the threshold for binarizing features; features less than or equal to the threshold are binarized to 0, and those greater than the threshold are binarized to 1.

These visual representations offer insights into the nuanced variations of hyperparameters across different Naive Bayes algorithms, providing a comprehensive understanding of their impact on model performance.

3.3.2 kNN

In the pursuit of optimizing the K-Nearest Neighbors (kNN) algorithm for bug severity classification, we conduct a comprehensive exploration of key hyperparameters using a random search. Figure 3.6 shows the variations and combinations of kNN hyperparameters. This figure serves as an illustrative example of the hyperparameter optimization process, where we search for the hyperparameters that lead to the best accuracy.

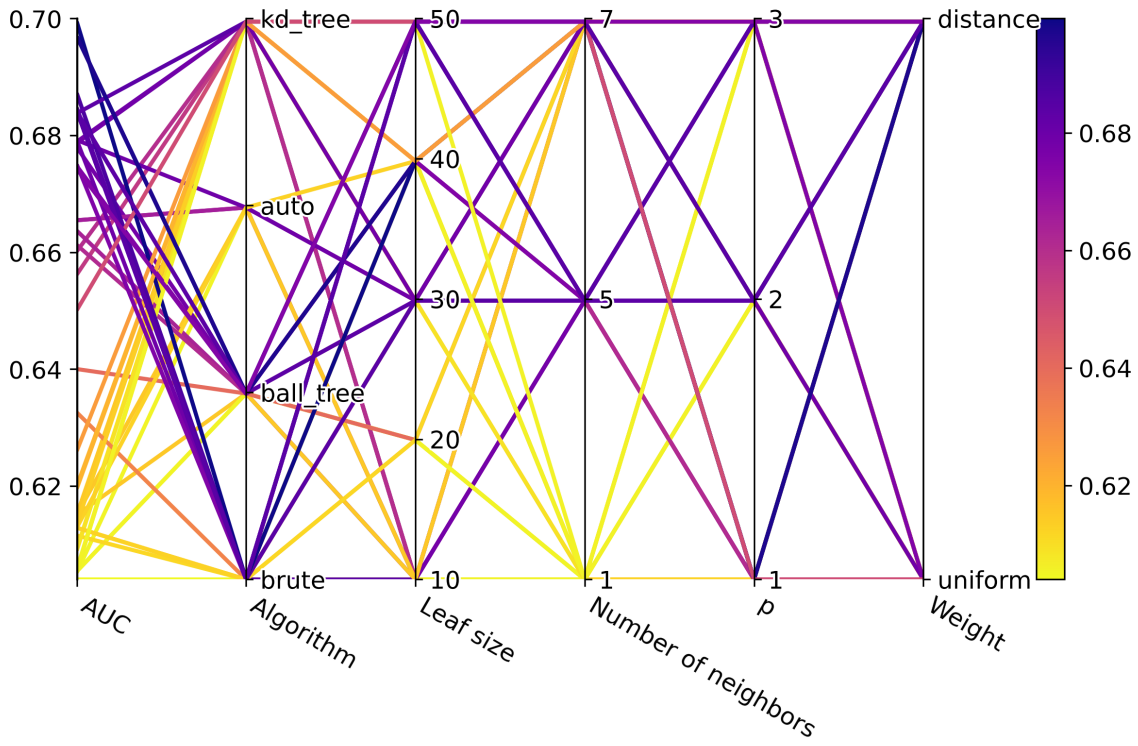


Figure 3.6 kNN hyperparameters with 1000 samples from the Mozilla dataset

During this process, various hyperparameters for the kNN algorithm are considered, covering the algorithm type, *leaf size*, *number of neighbors*, distance metric (p), and *weights*. The first y-axis in Figure 3.6, labeled AUC, represents the optimization metric. In this optimization endeavor, a total of 50 trials are conducted, each representing a unique attempt to fine-tune the algorithm and contribute to the robustness of our bug severity classification model.

Additionally, the *algorithm* hyperparameter, determining the method for calculating the nearest neighbor, undergoes variation across choices such as *kd_tree*, *auto*, *ball_tree*, and *brute*. We also explore the impact of *leaf size* and *number of neighbors* on the structure and efficiency of the kNN model. The *leaf size* hyperparameter, dictating the size of the leaf nodes in the algorithms, is explored across values like 10, 20, 30, 40, and 50. And the

number of neighbors is adjusted between 1, 5, and 7, influencing the size of the neighborhood considered during classification.

The p hyperparameter defines the type of distance used, with options including Manhattan ($p = 1$), Euclidean ($p = 2$), or Minkowski (arbitrary p) distances. The *weights* hyperparameter corresponds the weight function used in prediction, is explored for both *uniform* and *distance* options. Collectively, these hyperparameters influence how the kNN algorithm makes decisions, directly affecting its ability to identify patterns within the data.

3.3.3 SVC

We aim to optimize the Support Vector Classification (SVC) algorithm for bug severity classification. We conduct a hyperparameter exploration via random search. This process seeks the optimal combination of values to enhance predictive performance.

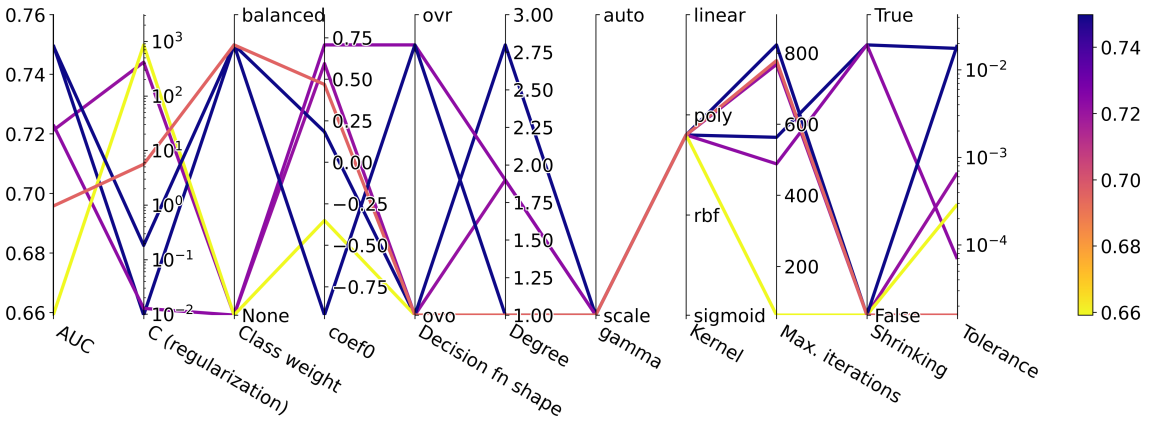


Figure 3.7 SVC hyperparameters with 1000 samples from the Mozilla dataset

Figure 3.7 is a parallel coordinates plot illustrating the hyperparameter optimization process for the SVC algorithm. Each line represents a unique combination of hyperparameter values, and the plot provides insights into the impact of varying parameters such as C, *kernel*, *gamma*, and others on the optimization metric (AUC).

The regularization parameter C in SVC critically modulates the trade-off between establishing a smooth decision boundary and accurately classifying training points. By adjusting C, practitioners control the width of the margin between decision boundaries. Smaller C values promote wider margins, potentially enhancing generalization but allowing more misclassifications. Conversely, higher C values yield smaller margins, directly impacting classification accuracy. Thus, the selection of C profoundly influences the SVC model's performance and its ability to balance between flexibility and precision in classification tasks. The variation

across a logarithmic scale from $1e-5$ to $1e5$ enabled us to systematically explore a broad range of regularization strengths.

The *class weight* hyperparameter provides a mechanism for adjusting the significance of various classes, addressing potential imbalances within the dataset. This functionality enables a more nuanced evaluation of each class's influence on the model's learning dynamics, offering options such as *balanced* or *None* to cater to different class distributions.

The *coef0* hyperparameter, relevant for *poly* and *sigmoid* kernels, allows us to fine-tune the independent term in the kernel function, influencing decision boundary placement. In our case, we vary *coef0* within the range of -1.0 to 1.

The *decision function shape* hyperparameter (Decision fn shape), a crucial aspect for the SVC algorithm, determines the format of the decision function returned by the model. In our experiments, we explore the two available options: *ovo* (one-vs-one) and *ovr* (one-vs-rest). The *ovo* decision function has a shape of (n_samples, n_classes * (n_classes - 1) / 2), representing a one-vs-one strategy. On the other hand, the *ovr* decision function follows the one-vs-rest strategy and has a shape of (n_samples, n_classes), consistent with decision functions in most classifiers. It is important to note that despite the user-specified format, internally, the *ovo* strategy is always used for training models, and the *ovr* matrix is constructed from the *ovo* matrix.

The choice of *kernel* function, guided by the *kernel* hyperparameter, significantly influence the decision boundary's shape. Options such as *linear*, *poly*, *rbf* (Radial Basis Function), and *sigmoid* provide diverse strategies for mapping input data into higher-dimensional spaces, impacting the model's ability to capture intricate patterns. In our random search, only the polynomial kernel was utilized, not by our preference, but rather due to the algorithm's selection for the random search process.

For scenarios where the *kernel* is set to *linear* or *rbf*, the *probability* hyperparameter is fixed at *True*, while its default value is *False*. This setting allows the model to estimate class probabilities, enriching the decision-making process.

Additionally, the hyperparameter *degree*, pertinent to polynomial kernels, exerts control over the polynomial degree of the kernel function. In our experimentation, we adjust the *degree* within the range of 1 to 5, allowing us to explore a spectrum of polynomial transformations.

The hyperparameter *gamma*, responsible for determining the kernel coefficient in *rbf*, *poly*, and *sigmoid*, is explored. The choices include specific values like *scale* and *auto*, providing flexibility in tailoring the influence of individual data points on the decision boundary.

The hyperparameter *max_iter* plays a crucial role in controlling the maximum number of

iterations for the convergence of the SVC algorithm. In our experimental setup, we introduce an additional binary hyperparameter, *is_max_iter*, to dynamically handle the behavior of *max_iter*. When *is_max_iter* is set to false, we assign -1 to *max_iter*, allowing the algorithm to continue until convergence. Conversely, when *is_max_iter* is true, we methodically vary *max_iter* within the range of 1 to 1000.

The *shrinking* hyperparameter controls the use of the shrinking heuristic, which aims to speed up the training process by removing support vectors that are unlikely to affect the decision boundary. In our hyperparameter optimization process, we explore the influence of *shrinking* by considering both *True* and *False* values. When set to *True*, the shrinking heuristic is enabled, potentially accelerating the training process by eliminating non-essential support vectors. Conversely, when set to *False*, the shrinking heuristic is disabled, resulting in a more exhaustive search for support vectors during training.

The hyperparameter *tol*, representing the tolerance for stopping criteria, undergoes variation across a logarithmic scale from $1e-5$ to $1e-1$. This range allow us to explore different levels of tolerance, influencing the convergence behavior of the SVC algorithm during training. A smaller tolerance value indicates stricter convergence criteria, potentially leading to longer training times but with the aim of achieving a more precise solution, while a larger tolerance allows for faster convergence but with a potentially less accurate result.

This exploration of hyperparameter space involves 50 trials for each configuration, aiming to understand the interaction between these parameters and their impact on the effectiveness of the *SVC* algorithm for bug severity classification.

3.3.4 Samples

We have a significant volume of data, consisting of 72192 samples from Eclipse (with 50534 for training, 14438 for validation, and 7220 for testing) and 199503 samples from Mozilla (with 139651 for training, 39901 for validation, and 19951 for testing). Our approach involves a randomized search for hyperparameters to optimize the models, necessitating the training, and validation of each model 50 times. However, conducting a comprehensive evaluation with the entire dataset proves time-consuming due to the inherent complexity of each algorithm.

In our pursuit of efficiency, and prompted by the hypothesis that truncating the dataset to a specific sample size does not significantly alter performance, we aimed to minimize search time. The objective is to find the number of samples that satisfy the training without influencing the prediction performance. To achieve this, we decided to truncate the training and validation datasets to sample sizes ranging from 200 to 8000 samples for hyperparameter

exploration. This decision is crucial for reducing the overall computational burden, as the time required for obtaining results is directly influenced by both the duration of model training and prediction tasks, which in turn depend on the number of samples used.

Moreover, longer tasks may lead to extended wait times in the processing queue, particularly if they require a significant amount of RAM. Therefore, finding the optimal balance between sample size and computational resources is essential for achieving efficient model optimization without compromising prediction accuracy.

Following the hyperparameter search, we identify the best model based on the AUC metric. Subsequently, we refine the training dataset to match the sample size associated with the best model. Unlike the hyperparameter search phase, the final evaluation encompasses the entire validation and test datasets.

The strategy of these steps enables us to strike a balance between computational times during hyperparameter search and the acquisition of accurate performance metrics.

3.4 Large Language Models

In the context of bug severity prediction using LLMs, we use different processes for inference and fine-tuning to predict severity levels, as depicted in Figure 3.1 at the outset of this chapter. For both processes, we use specific tools and configurations that will be discussed in the following subsections. We use PyTorch 3.10.2 and the HuggingFace Transformers library to implement the models. The LLM models chosen are `Llama2b-chat-hf` and `Llama-2-13b-chat-hf`. We opted for these models due to memory constraints encountered while attempting to employ the Llama2-70B model.

3.4.1 LLM Inference

LLM inference plays an essential role in bug severity prediction by leveraging the capabilities of Large Language Models. LLMs are advanced language models trained on vast amounts of text data, enabling them to understand and generate human-like text. In our context of bug severity prediction, LLMs are utilized for analyzing bug report descriptions and predicting whether reported bugs are severe. By processing the textual information provided in bug reports, LLMs can discern patterns and semantic cues indicative of the severity of the reported issues.

The flow of the approach using the pre-trained large language model is illustrated in Figure 3.8. We apply the Llama2-13B model in text generation mode, providing pre-processed

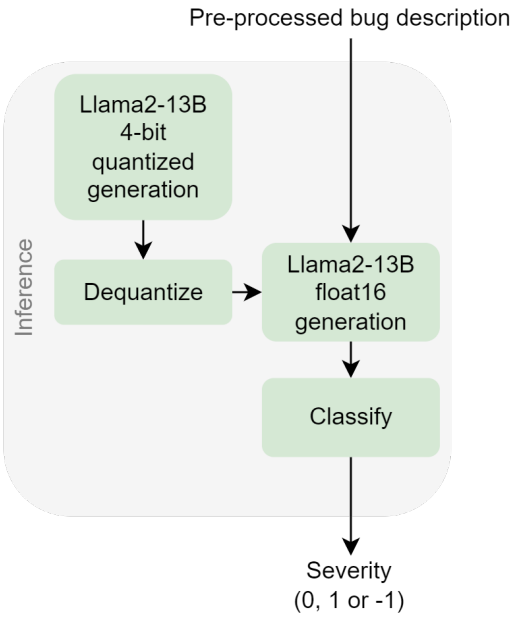


Figure 3.8 Inference pipeline: Bug severity prediction workflow using LLM pre-trained model

bug report descriptions as input prompts, as mentioned in section 3.2. To configure the model for inference, we use the `AutoModelForCausalLM` class from the HuggingFace Transformers library, which automatically selects a pre-trained causal language model based on the specified configuration.

The `AutoModelForCausalLM` class is designed to work with various transformer-based models that operate in a causal (left-to-right) context. By using this class, we can streamline the process of loading and initializing a pre-trained language model for text generation tasks. We also set specific parameters, such as quantization configuration, which uses the BitsAndBytesConfig class to apply double quantization for reduced memory usage, allowing us to optimize inference performance. Additional parameters, such as text generation settings (e.g., temperature, top-k, top-p), are configured to guide the model's output. We also map the model to the appropriate device, depending on available resources and hardware capabilities.

In Figure 3.8, pre-processed bug descriptions serve as input prompts to the Llama2-13B model. The model then generates text based on these prompts. This process involves two stages: initially, the model operates in a 4-bit quantized mode for efficient storage purposes. However, before generating text, when the weights of the model are needed for the forward or backward pass, the quantized representation is dequantized to float16 to ensure accurate text generation. Dequantization does not occur across the entire model; it is specific to the layer being utilized. Then the generated text is processed through a classify function, which interprets the model's outputs and assigns severity labels. These labels represent the

outputs of the bug severity prediction workflow, including 0 for non-severe bugs, 1 for severe bugs, and -1 for unclear responses. Unclear responses indicate instances where the model's outputs do not distinctly align with either non-severe (0) or severe (1) categorization. This methodology performs bug severity prediction by analyzing bug report descriptions using the frozen pre-trained Llama 2 model.

In the context of LLM inference, quantization and dequantization are important to optimize resource utilization. Quantization entails the conversion of the model's floating-point parameters into fixed-point representations with reduced precision, such as integers or lower-precision floating-point numbers. This reduction in precision enables more efficient storage and computation, which is particularly beneficial for hardware with limited resources. Moreover, quantization can enhance inference speed by reducing the computational complexity of the model. Conversely, dequantization involves reverting the quantized parameters back to their original floating-point representation when required for computation. This step is crucial for maintaining the accuracy of the model's predictions, as the precision reduction during quantization may introduce errors.

3.4.2 LLM Fine-tuning

Fine-tuning a pre-trained LLM involves adjusting its parameters on a specific dataset to enhance performance for a particular task. This section provides an in-depth exploration of the fine-tuning process, detailing the steps involved, hyperparameter selection, and the rationale behind our methodology. Through fine-tuning, we aim to optimize the LLMs' predictive capabilities and adapt them to handle bug severity prediction tasks effectively.

For the fine-tuning process, we utilize the Llama2-7B model in classification mode. Employing the `AutoModelForSequenceClassification` class from the HuggingFace Transformers library, which is designed to automatically select and initialize pre-trained transformer-based models for sequence classification tasks, such as text categorization or sentiment analysis.

This class provides an easy-to-use interface to create a model that includes a classification head, making it suitable for various supervised learning tasks. By employing the `AutoModelForSequenceClassification`, we streamline the configuration and setup process, allowing us to focus on fine-tuning the model to our specific needs.

To optimize resource usage during fine-tuning, we activate quantization with the `BitsAndBytesConfig` class, similar to the inference phase. This approach helps reduce memory consumption and computational overhead, which is crucial when working with large models like Llama2-7B. Additionally, we incorporate QLoRA, a specialized technique for low-rank adaptation, which

allows fine-tuning with minimal hardware requirements.

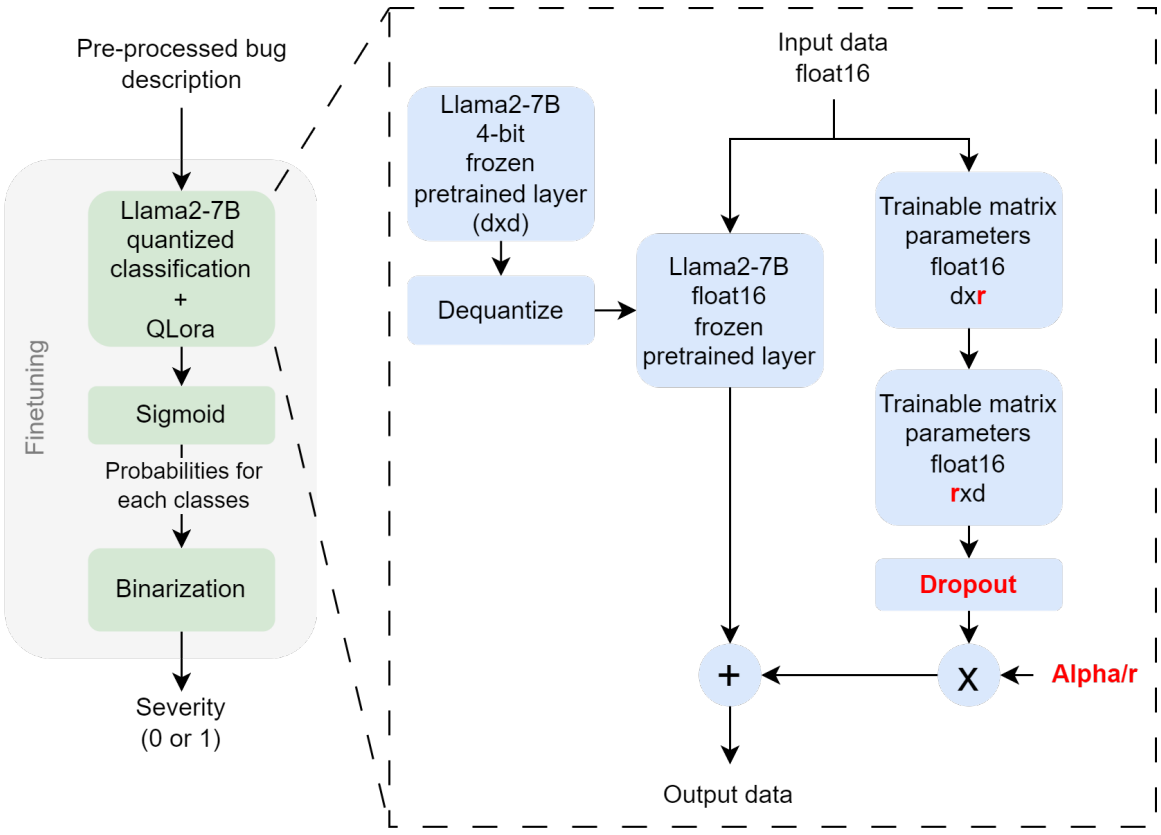


Figure 3.9 Fine-tuning pipeline: Bug severity prediction workflow using LLM pre-trained model and QLora

Figure 3.9 illustrates the operation of the Llama 2 model with QLora during the fine-tuning process. Similar to the previous approach, the input consists of pre-processed bug descriptions mentioned at the end of the section 3.2, and the dataset is divided into training, validation, and testing sets. The first green block in Figure 3.9 represents our Llama2-7B model, quantized in 4 bits, supplemented with QLora for fine-tuning.

The blue blocks in Figure 3.9 outline the steps involved when employing QLora during the fine-tuning process of our large pre-trained language model. The QLora technique freezes the Llama 2 weights at the start of fine-tuning. Initially, Llama2's weights are dequantized to the float16 format, matching the input data format. With this we can predict the output of the frozen pre-trained layer from the input data. For the LoRA part, we pass the input data through the two low-rank matrices. Subsequently, a dropout layer is applied to regularize the model. Finally, we combine the output of the LoRA adapter and the frozen pre-trained layer with a weighted sum, where the weight is determined by $\frac{\text{alpha}}{\text{r}}$.

As the fine-tuning progresses, layers are dequantized as needed, and backpropagation updates only the LoRA trainable matrices, leaving the frozen layers of the pre-trained model unchanged. This selective updating ensures that the pre-trained knowledge captured in the frozen layers remains intact while allowing the fine-tuning process to adapt the model's parameters to the specific task.

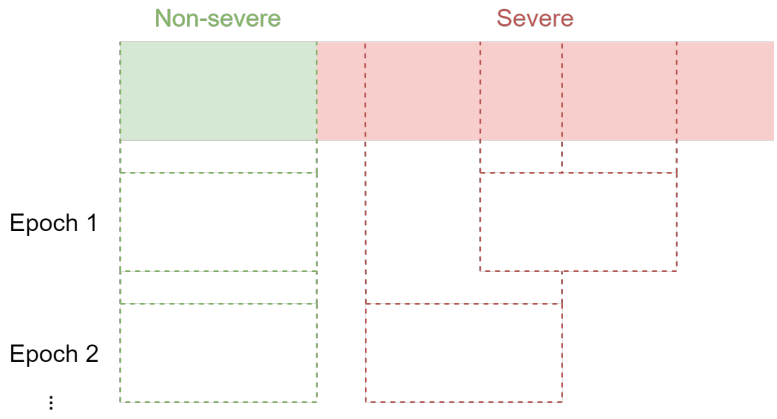


Figure 3.10 Random undersampling during training: For each epoch, all samples from the non-severe class, the majority class, are retained. An equal number of severe class samples are randomly selected from the majority class to balance the dataset.

To ensure balanced representation and improve training performance, we implemented dynamic subsampling of random data during the fine-tuning process. This method allows us to utilize all or almost all available samples from the majority class while maintaining equal representation of severe and non-severe samples. Figure 3.10 illustrates this process, where the non-severe class (green block) is preserved, and the range of the red block adjusts at each epoch to represent subsampled severe class data.

The Eclipse dataset contains 25% non-severe bugs, while the Mozilla dataset consists of 26% non-severe bugs. This imbalance poses a risk of biased model training, potentially leading to the neglect of important characteristics of the minority class. To address this, we retain 25% of non-severe bugs from the Eclipse dataset, equivalent to $12,664$ samples, and randomly sample from the severe class, totaling $37,870$ samples. Similarly, with the Mozilla dataset, which comprises 26% non-severe bugs ($36,433$ samples), we randomly sample from the severe class, consisting of $103,218$ samples.

Upon completion of the Llama 2 model's classification process, it generates a vocabulary-sized layer due to its training methodology, which focuses on generating text phrase by phrase. To transform this architecture into a classification model, we utilize the `AutoModelFor-`

`SequenceClassification` class from the HuggingFace Transformers library. This class adds a fully connected layer initialized with the desired number of outputs, enabling classification tasks. Additionally, we manually incorporate a sigmoid activation function to constrain the output values between 0 and 1.

In our implementation, this added layer facilitates the mapping of the vocabulary-sized Llama output representation to a single floating-point value. During fine-tuning, the model learns to associate the variations in this output with the probability of a bug description being classified as severe.

After this, we move to the *binarization* phase, which transforms the probabilities into 0 (non-severe) and 1 (severe), as illustrated in Figure 3.9. We selected the threshold yielding the highest validation accuracy to determine whether the result is 0 or 1.

Conducting a hyperparameter search with an LLM posed significant challenges due to the requirement for multiple GPUs with large VRAM capacities. To manage these constraints, we limited the search and computation time to 24 hours. Table 3.2 summarizes the hyperparameters investigated and their respective values tested via random hyperparameter search. Throughout the process, predictions on the full validation dataset are generated after each epoch, and checkpoints of the model are saved. Subsequently, the area under the curve (AUC) is computed for each hyperparameter combination tested at each epoch to identify the best-performing model. Once determined, the model weights are restored, and the model is evaluated on the training, validation, and test datasets.

Table 3.2 Hyperparameters used for finetuning Llama-7B and the values tested. Note that `weighted_sampling` refers to wether if we use random undersampling

Hyperparameters	**Domain**
`alpha`	$\{4, 10\}$
`dropout`	$\{0, 0.1, 0.2\}$
`r`	$\{5, 10, 64\}$
`learning_rate`	$\{1e-4, 1e-5\}$
`weighted_sampling`	$\{False, True\}$

The hyperparameters explored in this study are crucial for tuning the model's performance and behavior. The `alpha` parameter determines the scaling factor for the Low-Rank Adaptation (LoRa) matrices, impacting how aggressively the model adapts to the fine-tuning data. The `dropout` parameter helps prevent overfitting by randomly dropping out a fraction of the units during training, promoting generalization.

The `r` parameter specifies the rank of the LoRa matrices, influencing the dimensionality of

the adaptation. A higher rank allows for more complex adaptations but may increase computational requirements. The `learning_rate` parameter controls how quickly the model's weights are updated, affecting convergence speed and stability.

The `weighted_sampling` parameter addresses imbalances in the training data by undersampling the majority class, creating a balanced dataset during training. This strategy helps the model learn effectively from both majority and minority classes.

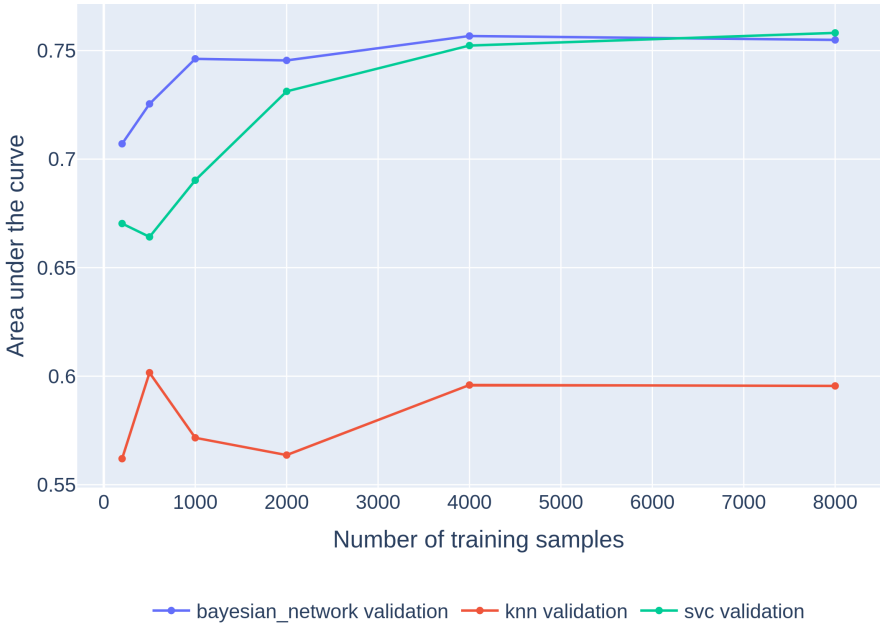
CHAPTER 4 RESULTS AND COMMENTS

This chapter presents the results of our bug severity prediction experiments using various machine learning approaches. We compare the performance of three baseline classifiers, namely Naive Bayes, k-Nearest Neighbors, and Support Vector Classifier, with that of Large Language Model approaches. Specifically, we investigate two LLM approaches: one utilizing the pre-trained Llama 2 model for inference and the other employing a fine-tuned Llama 2 model trained on our dataset. The evaluation metrics include confusion matrices, which provide insights into the classification performance of each approach, particularly in distinguishing between severe and non-severe bug reports. Through this comparative analysis, we aim to assess the effectiveness of LLMs in bug severity prediction tasks and determine the impact of fine-tuning on model performance.

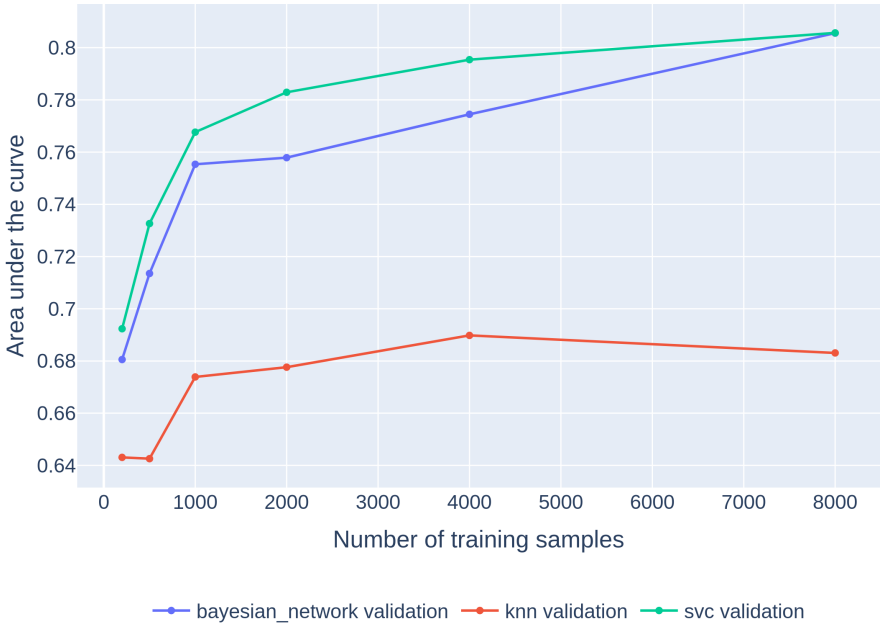
4.1 Baseline

This section presents the results of our bug severity prediction experiments using traditional machine learning approaches. We compare the performance of three baseline classifiers—Naive Bayes, k-Nearest Neighbors (kNN), and Support Vector Classifier (SVC)—with the aim of assessing their effectiveness in predicting bug severity. The evaluation metrics include the area under the receiver operating characteristic curve (AUC), and the confusion matrix. The experiments were conducted on two datasets—bug reports from Eclipse and Mozilla—with varying numbers of training samples to explore the impact of dataset size on classifier performance.

Figure 4.1 illustrates the relationship between the number of samples used in training the algorithms and the evaluation metric area under the curve (AUC), which indicates the model's performance in distinguishing between severe and non-severe bug reports. Each machine learning algorithm is represented by a different color, where the lines denote the AUC values on the validation data. It's important to note that a higher AUC value indicates better performance, with the AUC ranging from 0 to 1. Our approach involves a randomized search for hyperparameters to optimize the models, necessitating the training and validation of each model 50 times. However, conducting a comprehensive evaluation with the entire dataset proves time-consuming due to the inherent complexity of each algorithm. To address this challenge and streamline the experimentation process, we truncated the training and validation datasets to sample sizes ranging from 200 to 8000 samples for hyperparameter exploration. Our strategy aims to minimize the search time for hyperparameters and seeks



(a) Eclipse dataset



(b) Mozilla Dataset

Figure 4.1 Baselines

to find the number of training samples that maintain the performance of the algorithms without the need to use all available data for training.

Subfigure 4.1a illustrates the results obtained using bug reports from the Eclipse dataset. It

is evident that all three ML algorithms show an increase in performance up to 4000 samples, after which the AUC values stabilize. Specifically, k-Nearest Neighbors (kNN) stabilizes around 0.6 AUC, while the Bayesian network and Support Vector Classifier (SVC) stabilize around 0.75 AUC. However, a similar trend is not observed in Subfigure 4.1b, which represents the results using bug reports from the Mozilla dataset.

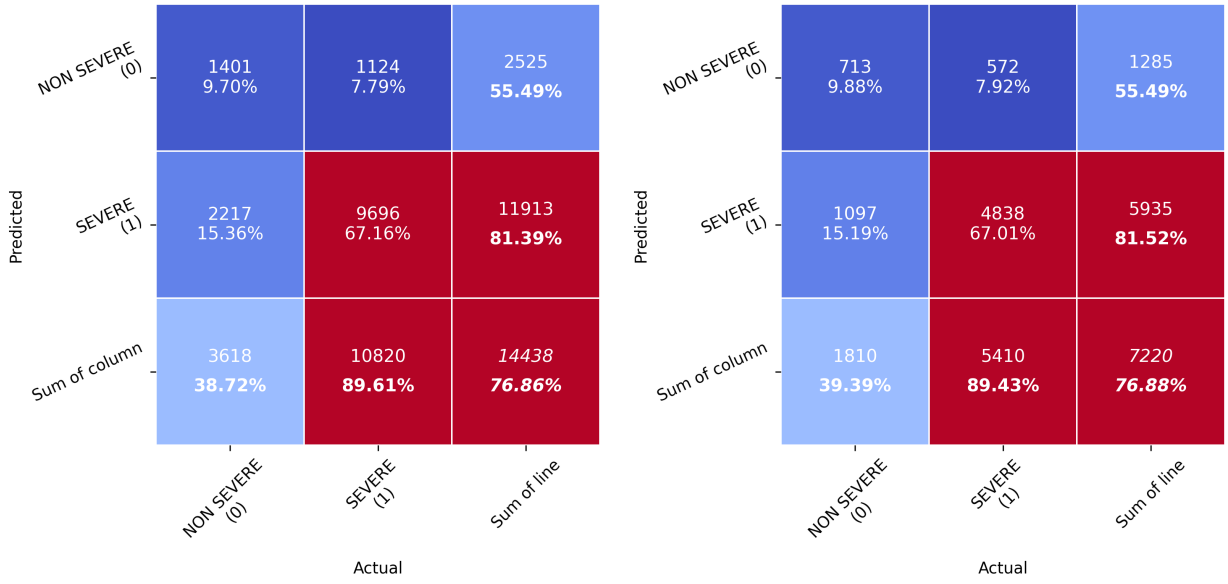
In the case of the Mozilla dataset, while kNN exhibits a peak performance with 4000 samples, the other two algorithms continue to show improved performance even with 8000 samples. This suggests that further increasing the sample size may lead to a normalization of the AUC gain. Additionally, the decision to refrain from increasing the number of samples for training in the Mozilla dataset was influenced by the time constraints to complete the experiments and present the findings in the report. Consequently, we could not reach a consensus on the optimal number of samples for the Mozilla dataset, and we chose to present only the Eclipse dataset results for comparison with the approaches using the Llama2 model.

Table 4.1 Hyperparameters used for Baselines with Eclipse dataset

Algorithm	**Hyperparameters**
Naive Bayes	classifier: Bernoulli
	`alpha`: 0.2754
	binarize: 0.3165
	Fit prior: `False`
kNN	Algorithm: `kd_tree`
	Leaf size: 30
	Number of neighbors: 7
	p: 3
	Weight: distance
SVC	C: 2.6849
	class weight: `balanced`
	coef0: −0.6584
	Decision function shape: `ovo`
	Degree: 1
	Gamma: `scale`
	Kernel: `poly`
	Max. iterations: −1
	Shrinking: `False`
	Tolerance: 0.0157
	Probability: `True`

Table 4.1 displays the hyperparameters obtained from the random search with a training and validation set of 4000 samples, chosen as it represents the point where the models converge

for the best AUC metric. Using these configurations and a seed equal to zero, we obtained the results presented in the confusion matrices of Figures 4.2, 4.3 and 4.4. Notably, the hyperparameter `Max. iterations` with a value of -1 indicates that there is no limit set; the algorithm will continue looping until it reaches the stopping criterion.



(a) Validation Confusion Matrix

(b) Test Confusion Matrix

Figure 4.2 Baselines: Confusion Matrices of Naive Bayes

The confusion matrices in Figure 4.2 provide a detailed breakdown of the classification performance of the Naive Bayes classifier for both the validation and test datasets. Each matrix consists of four quadrants, where the rows represent the predicted class (Output Class), and the columns represent the true class (Target Class). The diagonal cells indicate the number of correctly classified observations, while the off-diagonal cells represent incorrectly classified observations. Additionally, each cell displays both the count and percentage of the total number of observations.

To facilitate interpretation, we added a column to the right and a line below the matrix, which represents the sum of the observations in the rows and columns. The column on the far right of the plot displays the precision (or positive predictive value) for each class. Precision signifies the percentage of all examples predicted to belong to a particular class that is correctly classified. Similarly, the row at the bottom of the plot illustrates the recall (or true positive rate) for each class. Recall represents the percentage of all examples belonging to a class that is correctly classified.

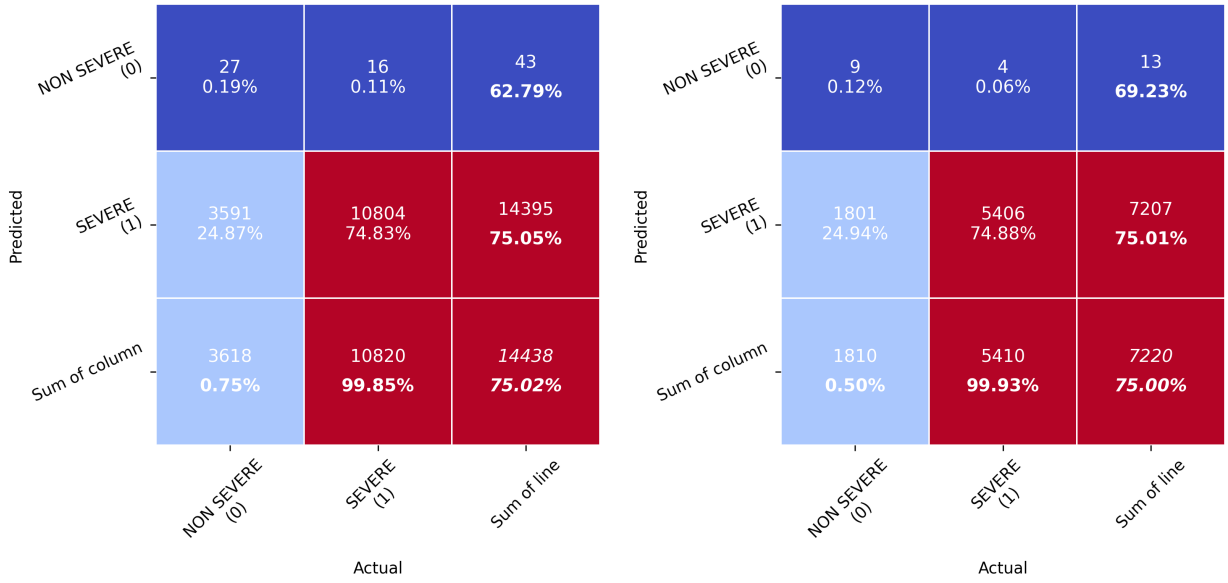
Furthermore, the bottom-right cell of the plot displays the overall accuracy, representing the percentage of all examples across both classes that are correctly classified. These metrics offer valuable insights into the classifier's performance and aid in assessing its effectiveness in bug severity prediction tasks. Moreover, the color intensity of each cell corresponds to its absolute value, with colors ranging from red to dark blue. Cells with higher values, such as 14438, appear in shades of red, while cells with lower values, such as 404, exhibit a darker shade of blue, indicating their comparatively lower impact.

For the validation confusion matrix of Naive Bayes in Figure 4.2a, the classifier correctly identified 1401 non-severe bug reports and 9696 severe bug reports. However, it misclassified 2217 non-severe bug reports as severe and 1124 severe bug reports as non-severe, resulting in a total of 3341 misclassified bug reports. This represents an overall precision of 55.49% for non-severe bug reports and 81.39% for severe bug reports. The recall rates are 38.72% for the non-severe class and 89.615% for the severe class. Also, the cell in the bottom right corner shows 76.86% of accuracy.

Similarly in Figure 4.2b, for the test dataset, the Naive Bayes classifier correctly classified 713 non-severe bug reports and 4838 severe bug reports. However, it misclassified 1097 non-severe bug reports as severe and 572 severe bug reports as non-severe, resulting in a total of 1669 misclassified bug reports. The classifier achieved a precision of 55.49% for non-severe bug reports and 81.52% for severe bug reports on the test dataset. Furthermore, the classifier demonstrated an overall accuracy of 76.88% on the test dataset.

The confusion matrices in Figure 4.3 provide a detailed breakdown of the classification performance of the k-Nearest Neighbors (kNN) classifier for both the validation and test datasets. For the validation confusion matrix of kNN in Figure 4.3a, the classifier correctly identified 27 non-severe bug reports and 10804 severe bug reports. However, it misclassified 3591 non-severe bug reports as severe and 16 severe bug reports as non-severe, resulting in a total of 3607 misclassified bug reports. This represents an overall precision of 62.79% for non-severe bug reports and 75.05% for severe bug reports. The recall rates are 0.75% for the non-severe class and 99.85% for the severe class. Also, the cell in the bottom right corner shows 75.02% of accuracy.

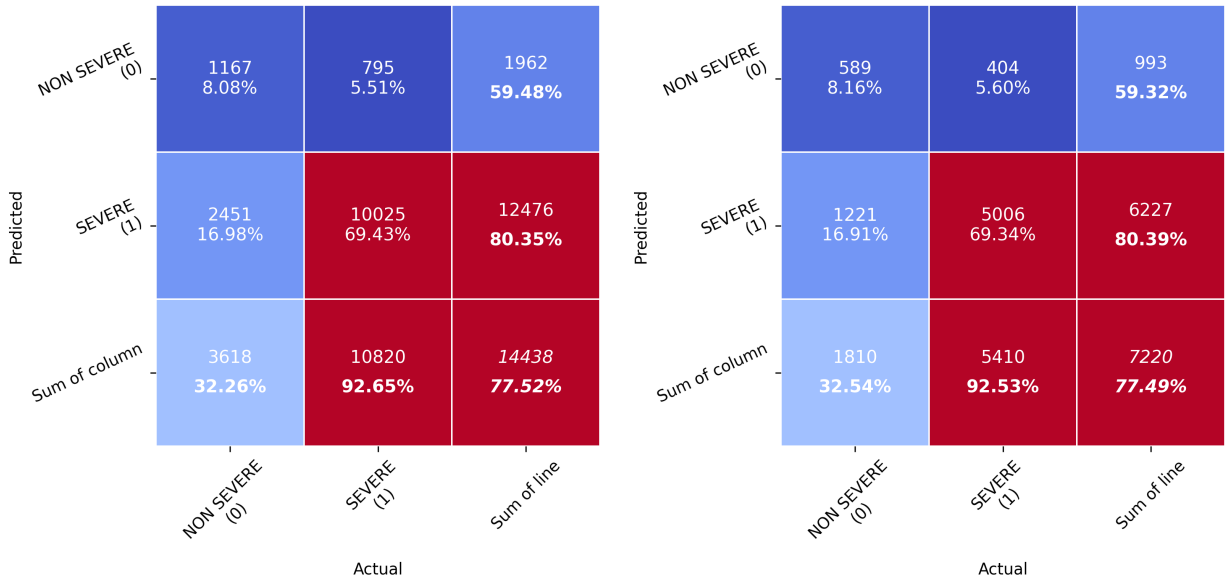
Likewise, for the test dataset (Figure 4.3b), the kNN classifier correctly classified 9 non-severe bug reports and 5406 severe bug reports. However, it misclassified 1801 non-severe bug reports as severe and 4 severe bug reports as non-severe, resulting in a total of 1805 misclassified bug reports. The classifier achieved a precision of 69.23% for non-severe bug reports and 75.01% for severe bug reports on the test dataset. And for the test, we have 75.00% of accuracy.



(a) Validation Confusion Matrix

(b) Test Confusion Matrix

Figure 4.3 Baselines: Confusion Matrices of kNN



(a) Validation Confusion Matrix

(b) Test Confusion Matrix

Figure 4.4 Baselines: Confusion Matrices of SVC

For the validation confusion matrix of SVC in Figure 4.4a, the classifier correctly identified 1167 non-severe bug reports and 10025 severe bug reports. However, it misclassified 2451

non-severe bug reports as severe and 795 severe bug reports as non-severe, resulting in a total of 3246 misclassified bug reports. This represents an overall precision of 59.48% for non-severe bug reports and 80.35% for severe bug reports. The recall rates are 32.26% for the non-severe class and 92.65% for the severe class. Additionally, the bottom right cell indicates an overall accuracy of 77.52%.

For the test dataset, in Figure 4.4b the SVC classifier correctly classified 589 non-severe bug reports and 5006 severe bug reports. However, it misclassified 1221 non-severe bug reports as severe and 404 severe bug reports as non-severe, resulting in a total of 1625 misclassified bug reports. The classifier achieved a precision of 59.32% for non-severe bug reports and 80.39% for severe bug reports on the test dataset. Furthermore, the recall rates are 32.54% for the non-severe class and 92.53% for the severe class. The accuracy for the test dataset is 77.49%.

Table 4.2 Accuracy of Baselines on Validation and Test Datasets with 4000 Samples

Alg	**Acc (%)**	
	Val	Test
NB	76.86	76.88
kNN	75.02	75.00
SVC	**77.52**	**77.48**

Note: All values are in percentages (%).

Table 4.3 Metrics of Baselines in Severe Class of Validation and Test Datasets with 4000 Samples

Alg	**Acc (%)**		**Prec (%)**		**Rec (%)**	
	Val	Test	Val	Test	Val	Test
NB	76.86	76.88	**81.39**	**81.53**	89.61	89.43
kNN	75.02	75.00	75.05	75.01	**99.85**	**99.93**
SVC	**77.52**	**77.48**	80.35	80.39	92.65	92.53

Note: All values are in percentages (%).

The tables 4.2, 4.3 and 4.4 summarize the accuracy, precision, and recall metrics in each class for the validation and test datasets, previously illustrated through confusion matrices. In terms of accuracy, the SVC algorithm outperforms others, exhibiting the highest performance in validation (77.52%) and test (77.52%) datasets. However, accuracy may provide a misleading evaluation in scenarios with imbalanced class distributions, such as the validation dataset with 3618 (25%) non-severe and 10820 (75%) severe bug reports, and the test dataset

Table 4.4 Metrics of Baselines in Non-Severe Class for Validation and Test Datasets with 4000 Samples

Alg	**Prec (%)**		**Rec (%)**	
	Val	Test	Val	Test
NB	55.49	55.49	**38.72**	**39.39**
kNN	**62.79**	**69.23**	0.75	0.50
SVC	59.48	59.32	32.26	32.54

Note: All values are in percentages (%).

with 1810 (25%) non-severe and 5410 (75%) severe bug reports. As accuracy measures the overall correctness of predictions, it may overemphasize the majority class (severe bug reports in this case) and mask the algorithm's performance on the minority class (non-severe bug reports).

Regarding precision and recall, considering the prediction of a bug report as severe is more critical than predicting it as non-severe due to the urgency of resolving issues. Therefore, in this analysis, we prioritize precision and recall when detecting severe bug reports. Precision is crucial when minimizing false positives is essential, ensuring that the identified severe bug reports are indeed severe. Conversely, recall is vital when minimizing false negatives, ensuring that all severe bug reports are correctly identified which is not the main objective.

In terms of precision, the Naive Bayes algorithm excels with 81.39% in validation and 81.53% in the test dataset for class severe. kNN demonstrates the best performance in recall measures, with rates of 99.85% for the validation dataset and 99.93% for the test dataset. However, it is crucial to note that kNN achieved the lowest recall rates in correctly identifying non-severe bug reports, with 0.75% for validation and 0.50% for the test, indicating that kNN classified almost all bug reports as severe.

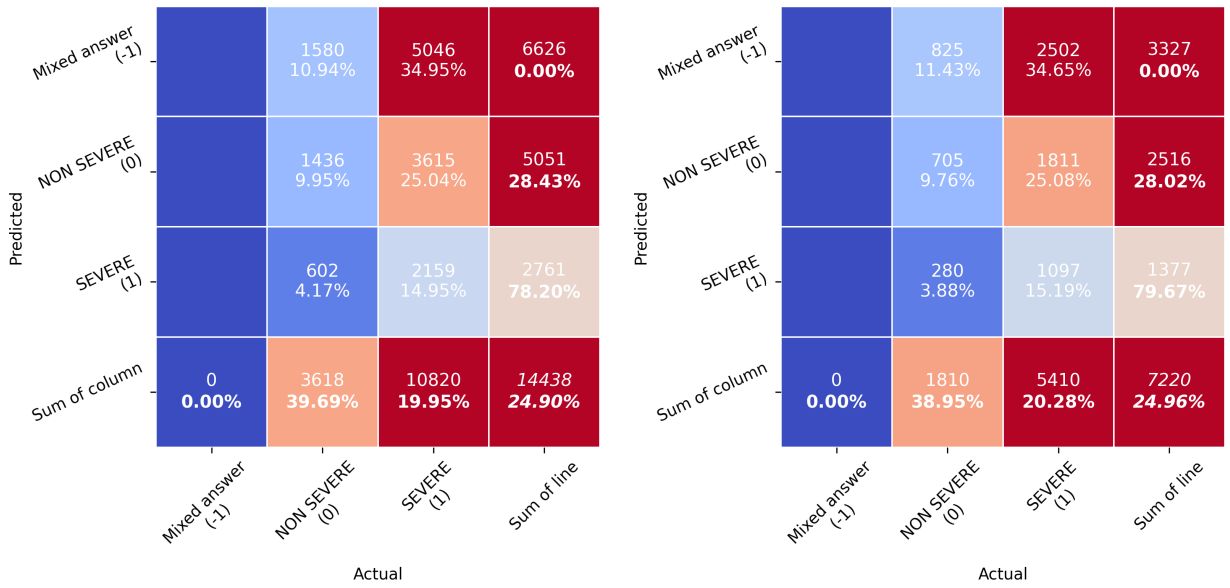
Therefore, while accuracy provides a general assessment of classification performance, precision offers more nuanced insights into the algorithm's ability to correctly identify severe bug reports. Recall, on the other hand, focuses on the ability to capture all relevant instances of the positive class, which is important but less critical in this context compared to precision. Given the urgency of resolving severe bug reports, prioritizing precision ensures that resources are allocated efficiently towards addressing the most severe issues.

4.2 Llama 2

This section explores the utility of Llama 2 for severity detection in bug reports, leveraging two distinct datasets: Eclipse and Mozilla data. Recognized for its adeptness in understanding and generating text, Llama2 presents a promising avenue for addressing this task. Consequently, we investigate its application, both by utilizing the pre-trained model and by fine-tuning it with our dataset. Additionally, we employ two prompt templates, the Official template provided by Meta, and Alpaca, a commonly used prompt template with LLMs. More details on these templates can be found in Appendix A.

4.2.1 Inference

The initial focus is evaluating Llama 2's inference capabilities when generating text based on prompts derived from the bug description of Eclipse and Mozilla datasets. By employing the pre-trained model, we aim to gauge its efficacy in understanding and responding to prompts tailored to these specific software development environments.



(a) Validation Confusion Matrix

(b) Test Confusion Matrix

Figure 4.5 Llama 2 Inference: Confusion Matrices of Eclipse Dataset with the Prompt Alpaca

Figure 4.5 presents the confusion matrices depicting the inference results obtained with the validation dataset and Eclipse tests using the Alpaca prompt, as outlined in Appendix A. It's noteworthy that the validation data serves for comparison with baseline and fine-tuning

outcomes. This separation of samples for training, validation, and testing was established beforehand and commented in Section 3.2. In Figure 4.5, classifications are categorized into non-severe, severe, and mixed answer, denoted by the value -1, indicating cases where the model's response does not align with either non-severe or severe categories.

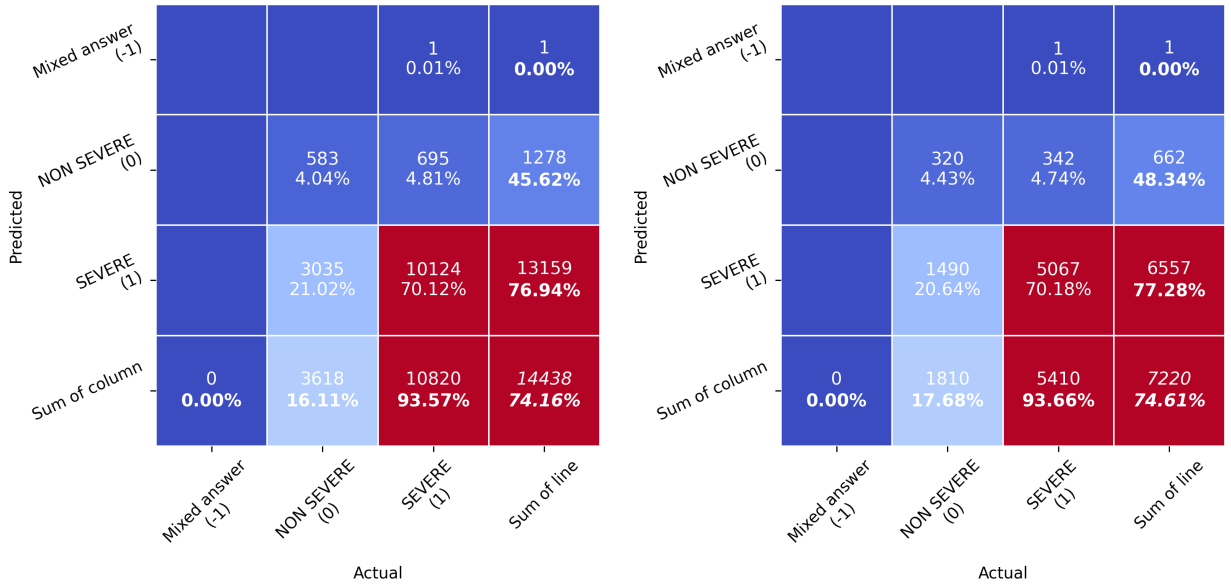
Correct predictions within the validation dataset manifest along the main diagonal of the matrix, with 1436 instances of non-severe samples (constituting 9.95% of the total) and 2159 severe samples (14.95% of the total). Similarly, 705 non-severe samples (9.76% of the total) and 1097 severe samples (15.19% of the total) were accurately predicted in the test dataset. Consequently, the accuracy for both datasets, utilizing the Alpaca prompt, hovers around 24.90%.

In the validation dataset, the accuracy stands at 20.43% for non-severe bug reports and 78.20% for severe bug reports. Notably, precision for severe bug reports accounts for instances where mixed answer values are disregarded, constituting 5046 samples (34.65% of the total) categorized as mixed answers. In this context, the total of true and false positives for precision are 2761 samples. Similar considerations apply to the test dataset, which exhibits an accuracy of 79.67% for severe bug reports, yet includes 2502 samples (34.65% of the total) predicted as mixed answers.

Furthermore, regarding recall, both non-severe and severe categories demonstrate relatively low values. For the validation dataset, recall rates are 39.69% for non-severe bug reports and 19.95% for severe bug reports. Likewise, in the test dataset, recall rates stand at 38.95% for non-severe bug reports and 20.28% for severe bug reports.

Figure 4.6 depicts the Eclipse bug report scenario, showcasing both the validation and test matrices generated using the official template for inference. This shift in templates provides notably divergent results. The correct predictions encompass 583 non-severe samples (4.04% of the total) and 10124 severe samples (70.12% of the total) in the validation dataset, and 320 non-severe samples (4.43% of the total) and 5067 severe samples (70.18% of the total) in the test dataset. Overall, the analysis reveals an accuracy of 74.16% for the validation dataset and 74.61% for the test dataset. Also, the recall metrics for severe bug reports have markedly improved, reaching 93.57% for the validation dataset and 93.66% for the test dataset.

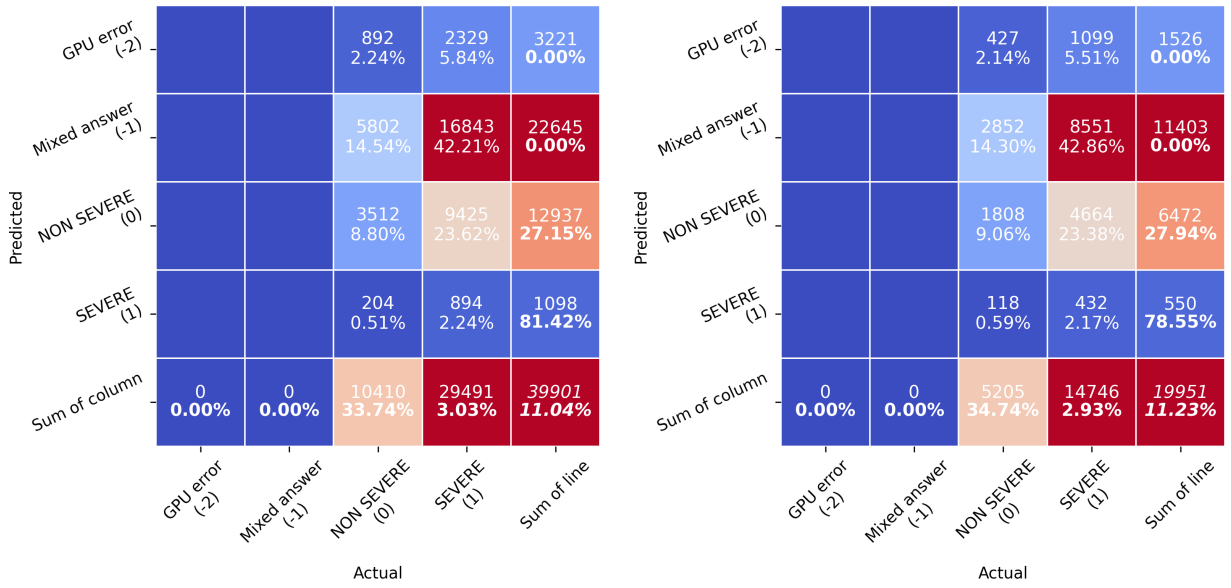
When comparing the severity detection accuracy values obtained using the official template with those from the Alpaca template, we observe similar accuracy rates, hovering around 77%. However, the official template yields only one sample categorized as a mixed answer, indicating a more refined prediction mechanism than the Alpaca template. Although the accuracy percentages remain consistent, the use of the official template results in a higher volume of accurately detected data.



(a) Validation Confusion Matrix

(b) Test Confusion Matrix

Figure 4.6 Llama 2 Inference: Confusion Matrices of Eclipse Dataset with the Prompt Official



(a) Validation Confusion Matrix

(b) Test Confusion Matrix

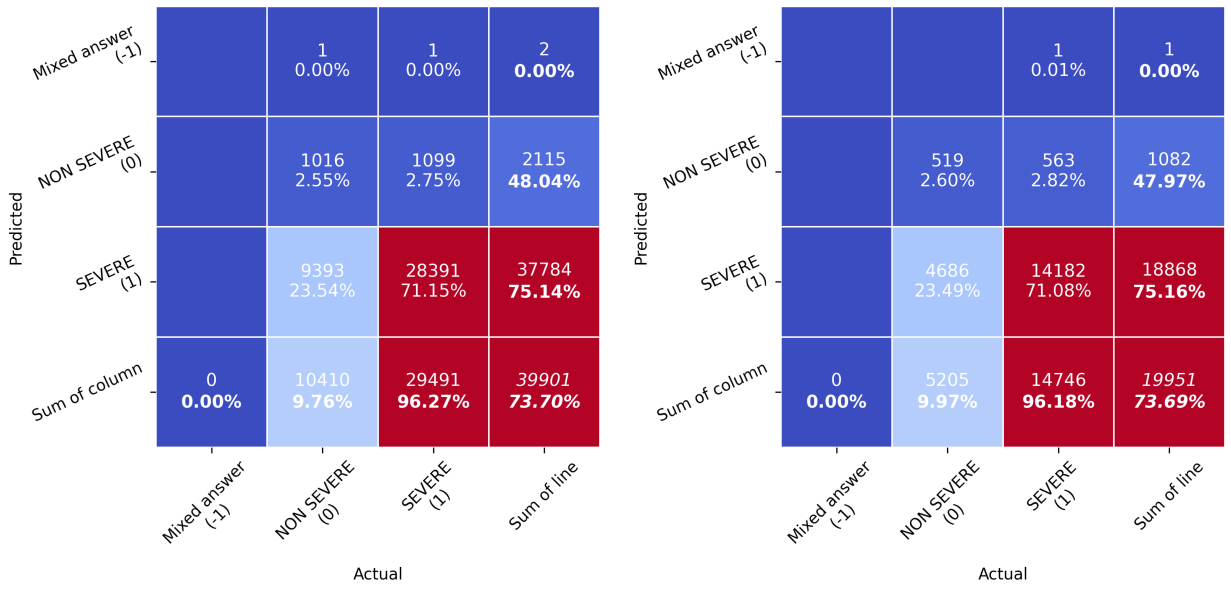
Figure 4.7 Llama 2 Inference: Confusion Matrices of Mozilla Dataset with the Prompt Alpaca

As observed with the Eclipse dataset, the Llama 2 model's performance using the Alpaca template also yielded suboptimal results when applied to the Mozilla dataset. Figure 4.7

illustrates these outcomes, revealing an accuracy of only 11.04% for the validation data and 11.23% for the test data. In the validation set, 3512 non-severe samples (8.80% of the total) and 894 severe samples (2.24% of the total) were correctly identified. Notably, a significant proportion of samples fall under the mixed answer category, indicating a lack of coherence in the model's predictions regarding the severity of bug reports. Specifically, mixed answer comprises 14.54% of non-severe samples and 42.21% of severe samples in the validation data.

Similar patterns are observed in the test dataset, where only 1808 non-severe samples (9.06% of the total) and 432 severe samples (2.17%) were correctly classified. High precision was achieved for the severe class due to the exclusion of mixed answer samples from the calculation. In this context, precision is computed based on the total of 1098 samples for validation and 550 samples for testing, encompassing true positives and false positives.

Moreover, using the Alpaca template with the Mozilla dataset, the inference process encountered additional challenges, including GPU errors. These errors occurred when attempting to infer specific samples, resulting in failed inference attempts for 3221 samples (8.07% of the total) for the validation dataset and 1526 samples (7.65% of the total) for the test dataset. Despite multiple attempts, no successful inferences were achieved for these samples, consistently resulting in GPU errors.



(a) Validation Confusion Matrix

(b) Test Confusion Matrix

Figure 4.8 Llama 2 Inference: Confusion Matrices of Mozilla Dataset with the Prompt Official

Figure 4.8 illustrates the confusion matrices generated from the Mozilla validation and test

datasets utilizing the Official template. Notably, this approach mitigates the issues encountered with GPU errors during inference, and the occurrence of Mixed answer samples is minimal, comprising only two samples for the validation data and one sample for the test data. Consequently, there is a notable increase in the number of samples correctly classified as severe, amounting to 28391 (71.15%) for the validation dataset and 14182 (71.08%) for the test dataset. However, the count of samples accurately identified as non-severe diminishes with the Official template, totaling 1016 samples (2.55% of the total) for validation and 519 samples (2.60% of the total) for testing.

Adopting the Official template yields an enhancement in accuracy, reaching around 73% for both Mozilla datasets. Notably, the model tends to classify bug reports as severe, which is evident from the substantial portion of non-severe bug reports predicted as severe, constituting around 23% of the dataset in both Mozilla datasets. Furthermore, precision and recall metrics for the severe class exhibit favorable outcomes, mirroring the performance observed with the Official template in the Eclipse dataset, boasting approximately 75% precision and 96% recall.

In summary, our evaluation highlights significant disparities between the Alpaca and Official templates when employing Llama 2 for inference. Across both the Eclipse and Mozilla datasets, the Alpaca prompt consistently yields inferior results compared to the Official template in terms of accuracy and recall for severe bug reports. Despite achieving remarkably high precision for severe bug reports, particularly with the Alpaca prompt, this is partly attributed to the dataset's imbalanced nature, with a larger proportion of severe bug reports. Additionally, the prevalence of samples labeled as "mixed answers" further complicates the interpretation of precision metrics.

Moreover, our experiments with the Mozilla dataset using the Alpaca prompt were hampered by GPU errors, hindering further analysis. Despite multiple attempts to provide the samples to the model, we were unable to achieve successful inference. This highlights potential limitations in Llama 2's ability to handle certain datasets or prompts effectively.

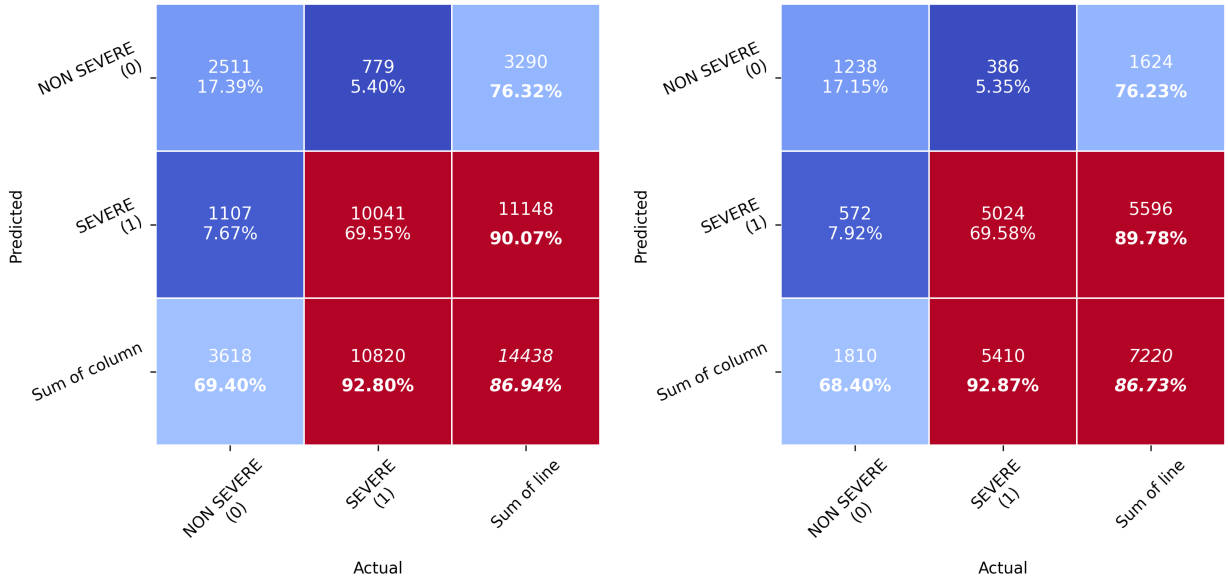
With the Official template, we observe a notable reduction in the occurrence of "mixed answers," aligning with expectations as it is the template used during Llama 2's training. While the Official template exhibits commendable precision, its precision values appear lower compared to the Alpaca prompt. However, this discrepancy can be attributed to the absence of "mixed answers" with the Official template, resulting in a higher volume of samples predicted as severe.

Overall, the accuracy consistently remains below 25% with the Alpaca prompt, indicating limitations in Llama 2's comprehension of bug descriptions. Conversely, accuracy surpasses

70% with the Official template, suggesting that while the model grasps the task at hand, its performance is constrained by its understanding of bug descriptions. These findings underscore the importance of prompt selection and dataset characteristics in leveraging Llama 2 for severity detection in bug reports.

4.2.2 Fine-tuning

In this section, we delve into the fine-tuning process of Llama 2, where we aim to enhance its performance in severity detection tasks using the Eclipse and Mozilla datasets. Fine-tuning involves adapting the pre-trained model to understand better and respond to prompts specific to bug report descriptions. We present the results obtained from fine-tuning Llama 2 using two different prompts: Alpaca and Official. The evaluation includes analysis of confusion matrices generated for both the validation and test datasets of each dataset, providing insights into the effectiveness of the fine-tuned models in accurately predicting bug severity.



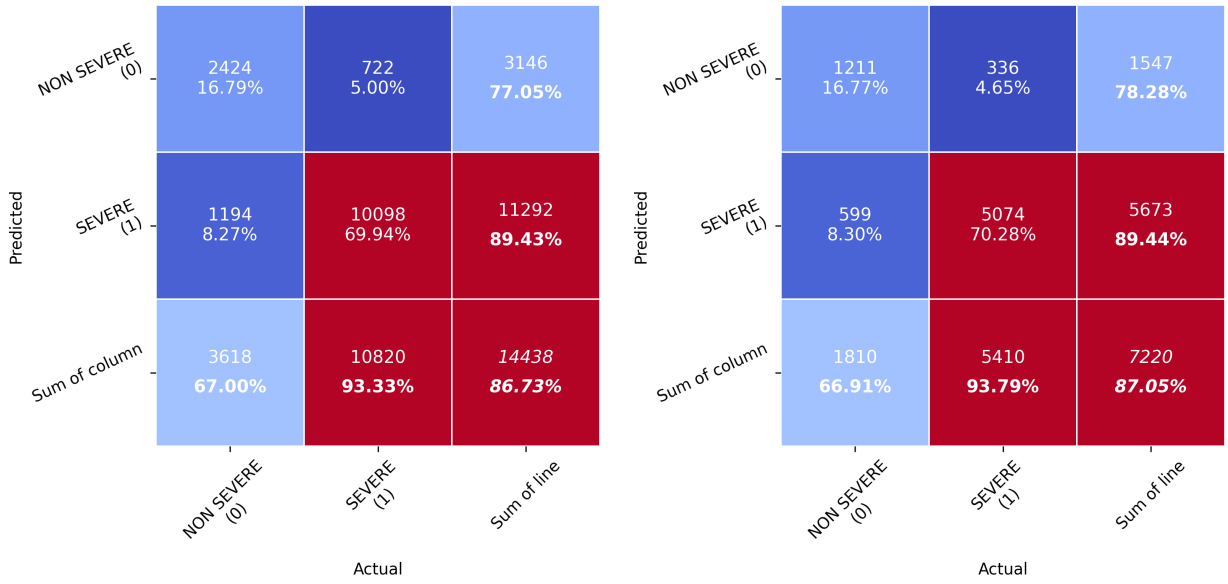
(a) Validation Confusion Matrix

(b) Test Confusion Matrix

Figure 4.9 Llama 2 Fine-tuning: Confusion Matrices of Eclipse Dataset with the Prompt Alpaca

Figure 4.9 illustrates the outcome of fine-tuning the Llama 2 model with bug report descriptions from the Eclipse dataset, utilizing the Alpaca prompt. The confusion matrices here only have the labels non-severe (0) and severe (1) because we are using the classification mode of the Llama 2 model.

In the validation dataset, depicted in Figure 4.9a, we observe 2511 samples classified as non-severe (constituting 17.39% of the total) and 10041 samples correctly predicted as severe (representing 69.55% of the total). Similarly, the test dataset, showcased in Figure 4.9b, demonstrates a comparable trend, with 17.15% and 69.58% of non-severe and severe samples correctly identified, respectively. Across both datasets, the model achieves an accuracy of approximately 86%, indicating its proficiency in distinguishing between severity levels. Moreover, the precision and recall metrics for the severe class consistently hover around 91%, underscoring the model's reliability in correctly identifying severe bug reports. Even the non-severe class exhibits relatively high precision and recall rates, standing at around 76% and 69%, respectively, across the datasets of Figure 4.9.



(a) Validation Confusion Matrix

(b) Test Confusion Matrix

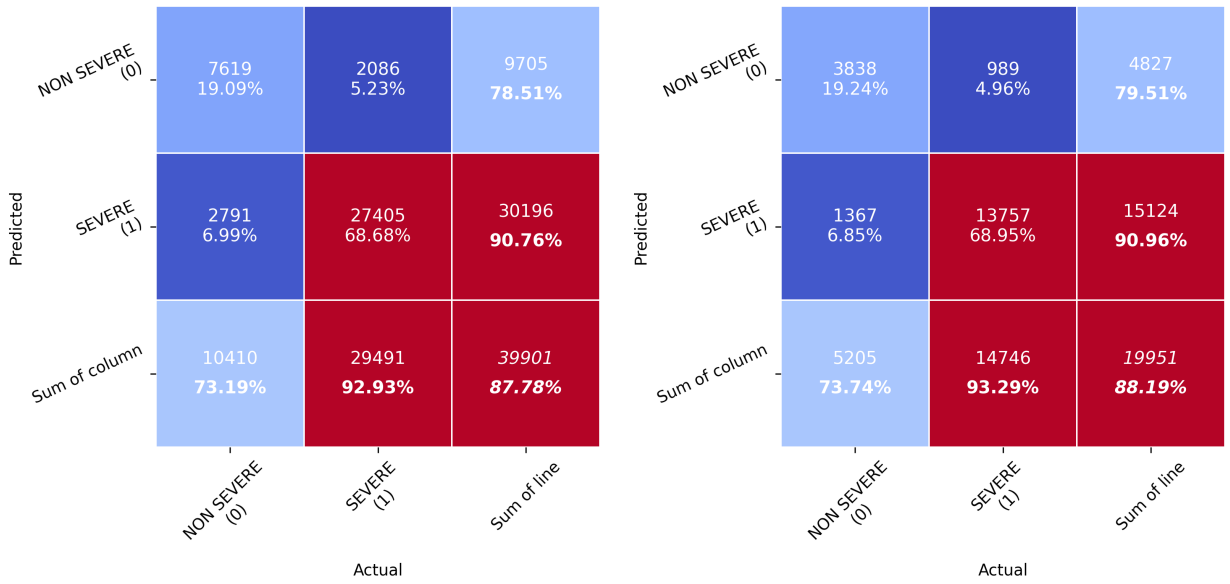
Figure 4.10 Llama 2 Fine-tuning: Confusion Matrices of Eclipse Dataset with the Prompt Official

The confusion matrices depicted in Figure 4.10 offer insights into the performance of the Llama 2 model fine-tuned with bug report descriptions from the Eclipse dataset, this time employing the Official prompt template. Despite the shift in prompt templates, the overall trends remain remarkably consistent with those observed in the Alpaca prompt model.

In both the validation and test datasets, approximately 16% of the total samples are accurately classified as non-severe, while around 70% of the total samples are correctly identified as severe. This consistency underscores the robustness of the model's ability to discern between severity levels, regardless of the prompt template used. Furthermore, the achieved

accuracy of around 87% signifies the model's proficiency in classifying bug reports with a high degree of accuracy.

Besides, we find that the precision and recall for the non-severe class hover around 77% and 67%, respectively, indicating a relatively balanced performance in correctly identifying non-severe bug reports. Conversely, for the severe class, the precision and recall metrics exhibit even higher levels of performance, standing at approximately 89% and 93%, respectively. These metrics highlight the model's efficacy in accurately identifying severe bug reports, important for prioritizing critical issues in software development workflows.



(a) Validation Confusion Matrix

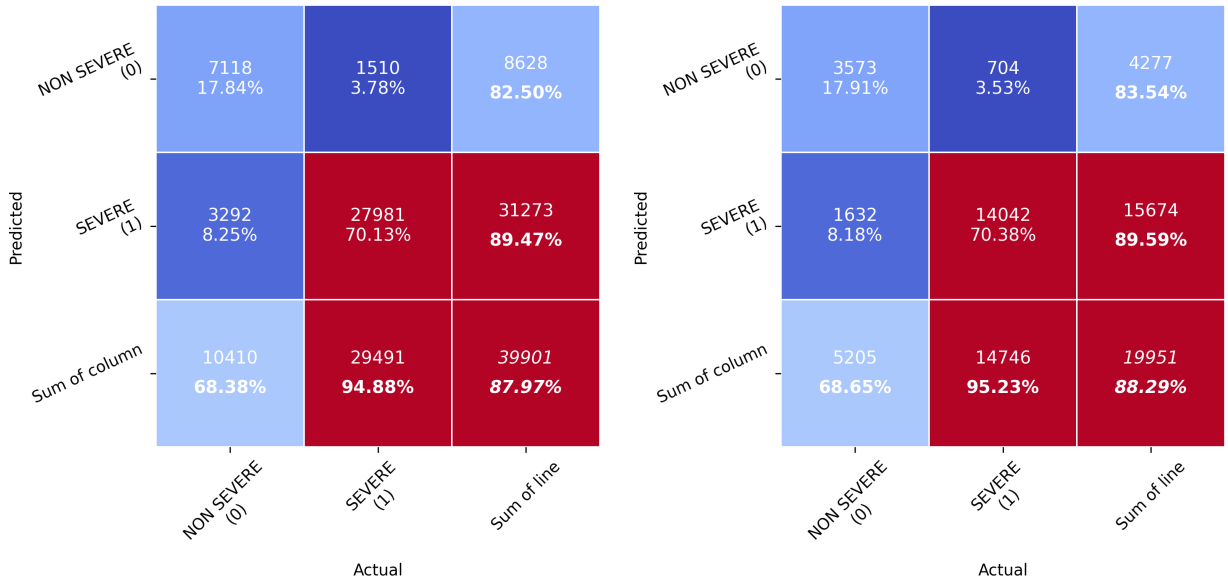
(b) Test Confusion Matrix

Figure 4.11 Llama 2 Fine-tuning: Confusion Matrices of Mozilla Dataset with the Prompt Alpaca

Figure 4.11 showcases the results obtained from fine-tuning the Llama 2 model with bug report descriptions from the Mozilla datasets, utilizing the Alpaca prompt template. The classification performance was observed in both the validation and test datasets.

Examining the confusion matrices for both the validation and test datasets reveals a consistent pattern, revealing the model's ability to classify bug reports into their respective severity categories accurately. Notably, approximately 19% of the total samples are correctly predicted as non-severe, while approximately 68% of the total samples are accurately identified as severe. These percentages underscore the model's proficiency in distinguishing between different severity levels, emphasizing correctly identifying severe bug reports.

Furthermore, the validation and test subsets exhibit an accuracy rate of 88%, reflecting the model's overall effectiveness in classifying bug reports with high precision. The precision and recall metrics hovered around 79% and 73%, respectively, for the non-severe class, indicating a balanced performance in correctly identifying non-severe bug reports. Similarly, the precision and recall metrics for the severe class stand at approximately 90% and 93%, respectively, underscoring the model's robustness in accurately identifying severe bug reports.



(a) Validation Confusion Matrix

(b) Test Confusion Matrix

Figure 4.12 Llama 2 Fine-tuning: Confusion Matrices of Mozilla Dataset with the Prompt Official

Figure 4.12 showcases the classification results obtained from fine-tuning the Llama 2 model using the Official prompt template with bug report descriptions from the Mozilla datasets. Upon examining the confusion matrices for both the validation and test datasets, a pattern emerges, similar to the observations made with the Alpaca prompt template. Approximately 17% of the total samples are correctly predicted as non-severe, while approximately 70% of the total samples are accurately classified as severe. These consistent percentages underscore the robustness of the model in accurately identifying bug severity levels across different subsets of the Mozilla dataset, irrespective of the prompt template used for fine-tuning.

Furthermore, both the validation and test subsets exhibit a commendable accuracy rate of 88%, reflecting the model's consistent performance in accurately classifying bug reports across different evaluation sets. The precision and recall metrics hovered around 82% and 68% for the non-severe class, indicating a balanced performance in correctly identifying non-

severe bug reports. Similarly, the precision and recall metrics for the severe class stand at approximately 89% and 94%, respectively, highlighting the model's effectiveness in accurately identifying severe bug reports.

4.2.3 Comparing Results: Inference and Fine-Tuning Approaches

This section explores the performance metrics obtained from inference and fine-tuning processes, shedding light on the model's ability to classify bug reports based on severity levels accurately. The tables 4.5, 4.6 and 4.7 present a comprehensive overview of the metrics obtained from the validation and test datasets for inference and fine-tuning approaches across the Eclipse and Mozilla datasets. The metrics include accuracy, precision, and recall for non-severe and severe bug reports.

Table 4.5 Accuracy of Validation and Test Datasets in Class Severe for Inference and Fine-tuning

	Alg	**Acc (%)**	
		Val	Test
Eclipse	Infer[1]	24.90	24.96
	Infer[2]	74.16	74.61
	Tune[1]	**86.94**	86.73
	Tune[2]	86.73	**87.79**
Mozilla	Infer[1]	11.04	11.23
	Infer[2]	73.70	73.69
	Tune[1]	87.78	88.19
	Tune[2]	**87.97**	**88.29**

Note: All values are in percentages (%).
[1] *Results using Alpaca prompt.*
[2] *Results using Official prompt.*

Analyzing the results reveals notable disparities between the inference and fine-tuning approaches, particularly in terms of accuracy and recall rates. In the Eclipse dataset, fine-tuning with the Alpaca prompt (Tune[1]) consistently outperforms inference (Infer[1]) in terms of accuracy, precision, and recall for both severity classes. This improvement underscores the efficacy of fine-tuning in enhancing the model's understanding of bug descriptions and its ability to classify bug reports based on severity levels accurately.

Similarly, fine-tuning with the Official prompt (Tune[2]) exhibits superior performance when compared to inference (Infer[2]) in the Eclipse dataset, thus reinforcing the critical role of prompt selection in enhancing the model's effectiveness. Notably, the fine-tuning process

Table 4.6 Metrics of Validation and Test Datasets in Class Severe for Inference and Fine-tuning

	Alg	**Prec (%)**		**Rec (%)**	
		Val	Test	Val	Test
Eclipse	Infer[1]	78.20	79.67	19.95	20.28
	Infer[2]	76.94	77.28	**93.57**	93.66
	Tune[1]	**90.07**	**89.78**	92.80	92.87
	Tune[2]	89.43	89.44	93.33	**93.79**
Mozilla	Infer[1]	81.42	78.55	3.03	2.93
	Infer[2]	75.14	75.16	**96.27**	**96.18**
	Tune[1]	**90.76**	**90.96**	92.93	93.29
	Tune[2]	89.47	89.59	94.88	95.23

Note: All values are in percentages (%).
[1] *Results using Alpaca prompt.*
[2] *Results using Official prompt.*

substantially improves various metrics, underscoring its efficacy in refining the model's understanding and response to bug report prompts. The exception lies in the recall metric for class severe, where the inference approach achieves a commendable 93.57% in the validation dataset, indicating its proficiency in accurately identifying severe bug reports even prior to fine-tuning.

Fine-tuning with both prompt models (Alpaca and Official) outperforms the inference metrics in most cases on the Mozilla dataset. The inference approach using the Official prompt template achieved a recall of 96.27% for class severe in the validation dataset, demonstrating the model's high proficiency in correctly identifying severe bug reports even before fine-tuning. Similarly, in the test dataset, the recall for class severe was 96.18%, reaffirming the model's consistent performance in identifying severe bug reports accurately.

Comparing the inference performance of pre-trained models across different prompt templates unveils intriguing insights. Notably, utilizing the Official template consistently yielded significantly higher metrics compared to the Alpaca template for both datasets. This discrepancy may be attributed to the Official template being the one used during Llama 2's training by Meta, suggesting a closer alignment between training and inference conditions. However, in the case of fine-tuning, both prompt templates yielded good results with closely matched metrics.

Furthermore, the comparison between inference and fine-tuning approaches underscores the effectiveness of fine-tuning in enhancing Llama 2's performance in severity detection tasks.

Table 4.7 Metrics of Validation and Test Datasets in Class Non-Severe for Inference and Fine-tuning

	Alg	**Prec (%)**		**Rec (%)**	
		Val	Test	Val	Test
Eclipse	Infer[1]	28.43	28.02	39.69	38.95
	Infer[2]	45.62	48.34	16.11	17.68
	Tune[1]	76.32	76.23	**69.40**	**68.40**
	Tune[2]	**77.05**	**78.28**	67.00	66.91
Mozilla	Infer[1]	27.15	27.94	33.74	34.74
	Infer[2]	48.04	47.97	9.76	9.97
	Tune[1]	78.51	79.51	**73.19**	**73.74**
	Tune[2]	**82.50**	**83.54**	68.38	68.65

Note: All values are in percentages (%).
[1] *Results using Alpaca prompt.*
[2] *Results using Official prompt.*

Notably, there were instances where the inference outperformed fine-tuning, particularly in the Mozilla dataset. This observation suggests that the limited training time, constrained by a 24-hour timeout and the inability to fully exploit the training potential, may have contributed to suboptimal fine-tuning results. A hypothesis emerges that prolonged training could yield better fine-tuning performance, warranting further investigation.

Another hypothesis for the three cases where inference obtained better results than fine-tuning is the inherent risks associated with fine-tuning large language models. Fine-tuning involves exposing the model to specific datasets and prompts, with the aim of adapting its parameters to better fit the task at hand. However, fine-tuning such models comes with the risk of catastrophic forgetting, where the model's ability to retain previously learned information is compromised as it adapts to new data. The observed limitations in fine-tuning performance may stem from the model overfitting to the training data or failing to generalize well to unseen examples.

In conclusion, fine-tuning the Llama 2 model using both the Alpaca and Official prompt templates showcases promising results in bug severity prediction tasks across the Eclipse and Mozilla datasets. This adaptation of the model presents a valuable approach for enhancing bug management processes in software development, offering insights into the model's capabilities and potential avenues for further refinement.

4.3 Performance Comparison: Baselines versus Llama 2

In this section, we compare the performance of baseline machine learning algorithms with that of Llama 2 on the Eclipse dataset. The evaluated metrics include accuracy, precision, and recall for both non-severe and severe bug reports.

Table 4.8 Accuracy for Eclipse Dataset: Baselines with 4000 Samples of training, Inference, and Fine-Tuning

Metric	**Dataset**	**NB**	**kNN**	**SVC**	**Infer**[1]	**Infer**[2]	**Tune**[1]	**Tune**[2]
Accuracy (%)	Val	76.86	75.02	77.52	24.90	74.16	**86.94**	86.73
	Test	76.88	75.00	77.48	24.96	74.61	86.73	**87.79**

Note: All values are in percentages (%).
[1] *Results using Alpaca prompt.*
[2] *Results using Official prompt.*

Table 4.9 Metrics for Severe Class in Eclipse Dataset: Baselines with 4000 Samples of training, Inference, and Fine-Tuning

Metric	**Dataset**	**NB**	**kNN**	**SVC**	**Infer**[1]	**Infer**[2]	**Tune**[1]	**Tune**[2]
Precision (%)	Val	81.39	75.05	80.35	78.20	76.94	**90.07**	89.43
	Test	81.53	75.01	80.39	79.67	77.28	**89.78**	89.44
Recall (%)	Val	89.61	**99.85**	92.65	19.95	93.57	92.80	93.33
	Test	89.43	**99.93**	92.53	20.28	93.66	92.87	93.79
F1-Score (%)	Val	85.30	85.69	86.06	31.79	84.44	**91.41**	91.34
	Test	85.30	85.63	86.03	32.33	84.69	91.30	**91.56**

Note: All values are in percentages (%).
[1] *Results using Alpaca prompt.*
[2] *Results using Official prompt.*

The tables 4.8, 4.9 and 4.10 present the results obtained from the evaluation of various algorithms. The baselines include Naive Bayes (NB), k-Nearest Neighbors (kNN), and Support Vector Classifier (SVC), while Llama 2 is assessed in two modes: inference using prompts generated from the Alpaca and Official templates (Infer[1] and Infer[2]), and fine-tuning with the same templates (Tune[1] and Tune[2]).

Among the baseline algorithms, Naive Bayes achieves an accuracy of 76.86% on the validation set and 76.88% on the test set. k-Nearest Neighbors and Support Vector Classifier also demonstrate competitive performance, with accuracies ranging from 75% to 77.52% on the validation set and test set.

Table 4.10 Metrics for Non-Severe Class in Eclipse Dataset: Baselines with 4000 Samples of training, Inference, and Fine-Tuning

Metric	**Dataset**	**NB**	**kNN**	**SVC**	**Infer**[1]	**Infer**[2]	**Tune**[1]	**Tune**[2]
Precision (%)	Val	55.49	62.79	59.48	28.43	45.62	76.32	**77.05**
	Test	55.49	69.23	59.32	28.02	48.34	76.23	**78.28**
Recall (%)	Val	38.72	0.75	32.26	39.69	16.11	**69.40**	67.00
	Test	39.39	0.50	32.54	38.95	17.68	**68.40**	66.91
F1-Score (%)	Val	45.61	1.48	41.83	33.13	23.81	**72.70**	71.67
	Test	46.07	0.99	42.03	32.59	25.89	72.10	**72.15**

Note: All values are in percentages (%).
[1] *Results using Alpaca prompt.*
[2] *Results using Official prompt.*

In comparison, Llama 2 exhibits varying performance levels depending on the operation mode. During inference, Llama 2 achieves accuracies of 24.90% to 74.16%, with higher values observed in the fine-tuning mode, ranging from 86.73% to 86.94%. Notably, fine-tuning with the Official prompt (Tune[2]) yields the highest accuracy of 87.79% on the test set.

Regarding precision and recall, Llama 2 generally outperforms the baseline algorithms, particularly in the detection of severe bug reports. For instance, fine-tuning with the Official prompt (Tune[2]) achieves a recall of 93.79% for class severe on the test set, surpassing the recall values of the baseline algorithms.

When considering the F1-Score metric, which combines precision and recall values using a harmonic mean, we observe that the highest values are achieved by fine-tuning the LLMs (Tune[1] and Tune[2]). Concerning the validation data, the Alpaca template outperforms the Official template, with scores of 91.41% for class severe, and 72.70% for class non-severe. Conversely, concerning the test data, we observe the opposite trend, with the Official template yielding the best results, scoring 91.56% and 72.15% for class severe and class non-severe respectively. However, it is worth noting that the difference between the F1-Score values of the fine-tuned LLMs is small.

These findings suggest that Llama 2, especially when fine-tuned with appropriate prompts, holds promise for bug severity detection tasks in software development contexts. The superior performance of Llama 2 underscores the potential of large language models in automating and enhancing software quality assurance processes.

CHAPTER 5 CONCLUSION

In this concluding chapter, we offer a succinct overview of our study's key findings, followed by the conclusive insights derived from our experimental outcomes. Additionally, we delve into the inherent limitations encountered during our research and outline potential avenues for future exploration aimed at addressing aspects not covered within the scope of this study.

5.1 Summary of Works

Severity prediction of bug reports is a crucial task in software engineering, as it aids in prioritizing and addressing critical issues efficiently. While classical algorithms have been traditionally used for this purpose, recent advancements in natural language processing (NLP) have introduced the possibility of leveraging large language models (LLMs) for more accurate predictions.

We explore the application of LLMs, specifically Llama 2, for severity prediction of bug reports using the Eclipse and Mozilla datasets. We compare the performance of Llama 2 with baseline algorithms, including Naive Bayes, k-Nearest Neighbors, and Support Vector Classifier, on these datasets.

Our approach involves both inference with pre-trained Llama 2 models and fine-tuning Llama 2 with custom prompts tailored to bug report severity prediction. Despite encountering hardware limitations, such as GPU constraints necessitating the use of QLora, we observed promising results with Llama 2, demonstrating its efficacy in bug severity prediction tasks.

While the baseline algorithms achieve accuracies ranging from 75% to 81% for the severe class (class 1) on the Eclipse dataset, our fine-tuned Llama model demonstrates an accuracy of 90%. The kNN algorithm exhibits impressive recall values for the severe class; however, upon closer examination, it becomes apparent that this is largely due to an imbalance in predictions, with almost the entire dataset classified as severe, resulting in recall values of 0.75 and 0.5 for the non-severe class (class 0). In contrast, the fine-tuned Llama models achieve a balanced recall of approximately 92% for class 1 and 68% for class 0, indicating robust performance across both severity categories. Moreover, when considering the F1-Score metric, the fine-tuned LLMs achieve the highest values, ranging from 71.67% to 72.10% for class 0 (non-severe) and from 91.30% to 91.56% for class 1 (severe). In contrast, the baselines have the F1-Score values from 0.99% to 46.07% for class 0 and from 85.30% to 86.06% for class 1.

5.2 Limitations and Future Research

Although our study has yielded promising results, it has also encountered several limitations that merit careful consideration and highlight areas for future investigation. Chief among these limitations is the constraint imposed by hardware, particularly the availability of GPUs, which presented significant challenges in scaling our experiments and may have affected the robustness of our findings. Concurrent access to multiple A100 GPUs for fine-tuning was impractical, prompting us to leverage QLora to alleviate memory requirements during the fine-tuning process[1]. While QLora's use was beneficial in mitigating hardware demands, it came with the drawback of being unable to utilize more than one GPU. Nevertheless, this compromise was necessary to circumvent the potential issues from excessive hardware demands. Moreover, it's crucial to note that aside from the memory required for the model, a GPU memory allocation was necessary for processing the data, leading to the truncation of bug report descriptions during data pre-processing.

The hardware limitation necessitated imposing a 24-hour time limit on the fine-tuning of the Llama 2 model to speed up results acquisition. For instance, our parameter search involved initiating multiple parallel jobs, resulting in lower priority in the Compute Canada job queue. Consequently, we experienced some delays that could reach approximately a week before new jobs could commence. In response, we implemented measures to accelerate the process, including imposing time constraints on individual jobs.

Testing other Large Language Models of varying sizes presents an intriguing avenue for further research. With the emergence of new open-source LLMs like Mixtral, exploring their performance in bug severity prediction tasks could offer valuable insights. Assessing the interpretability of different LLMs and their implications for bug severity prediction represents another promising direction. Interpretability aims to make LLMs more transparent by providing insights into the features, patterns, or linguistic cues that influence their predictions. This transparency is crucial, especially in critical applications like bug severity prediction, where stakeholders need to understand why a particular bug report is classified as severe or non-severe.

Expanding the binary classification of bug severity into multiple levels, such as trivial, minor, major, critical, and blocker, aligns with industry standards and could enhance the granularity of severity predictions. Additionally, incorporating bug reports labeled as normal or improvement introduces a new dimension to severity prediction analysis. Comparing model performance when including or excluding such data, both for LLMs and traditional algo-

[1] https://huggingface.co/spaces/Vokturz/can-it-run-llm

rithms, would shed light on the impact of these labels on prediction accuracy. Despite the common practice of removing normal and improvement labels due to noise or perceived insignificance, empirical evidence quantifying their influence on bug severity prediction remains scarce in the literature.

One significant area for future research is the analysis of inference time for bug severity prediction across different Large Language Models (LLMs). Inference time plays a critical role in determining the practicality of deploying these models in real-world settings, particularly in scenarios where rapid bug triage is crucial. Given the computational complexity of LLMs, understanding their performance in terms of response time can offer valuable insights into their suitability for operational use. Furthermore, inference time can vary significantly between different models, even if their accuracy or other performance metrics are comparable.

To address this, future research could compare inference times for bug severity prediction between various LLM architectures and configurations, such as those employing fine-tuning with QLora versus standard fine-tuning, or between smaller and larger LLMs. This analysis could help identify the trade-offs between model accuracy and response time, providing a basis for selecting the most appropriate model for different contexts, such as development environments or production systems. Moreover, the influence of hardware constraints on inference time, such as GPU availability and memory limitations, should be further investigated to understand how they impact the overall performance of LLM-based severity prediction.

Additionally, exploring inference time from a user perspective could offer practical insights into the user experience and acceptance of automated severity prediction tools. This could involve conducting user studies to gauge the impact of inference delays on developer workflows and whether faster response times lead to improved productivity or user satisfaction.

Overall, our study lays the groundwork for future research efforts that aim to leverage large language models for automated bug severity prediction and advance software engineering practices. By addressing the identified limitations and exploring avenues for further investigation, researchers can continue to enhance the accuracy, scalability, and interpretability of severity prediction models. Additionally, our findings underscore the importance of collaboration between academia and industry to overcome hardware constraints, model interpretability, and data preprocessing challenges. Through continued exploration and innovation, we can harness the full potential of large language models to improve bug management processes and ultimately enhance software quality and reliability.

REFERENCES

[1] L. A. F. Gomes, R. da Silva Torres, and M. L. Côrtes, "Bug report severity level prediction in open source software: A survey and research opportunities," *Information and Software Technology*, vol. 115, pp. 58–78, 11 2019. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0950584919301648

[2] Eclipse Foundation, "Eclipse Bug Tracking," https://wiki.eclipse.org/Eclipse/Bug_Tracking, accessed on: January 13, 2024.

[3] A. Lamkanfi *et al.*, "Predicting the severity of a reported bug," *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pp. 1–10, 5 2010. [Online]. Available: http://ieeexplore.ieee.org/document/5463284/

[4] ——, "Comparing mining algorithms for predicting the severity of a reported bug," *2011 15th European Conference on Software Maintenance and Reengineering*, pp. 249–258, 3 2011. [Online]. Available: http://ieeexplore.ieee.org/document/5741332/

[5] G. Yang *et al.*, "Analyzing emotion words to predict severity of software bugs," *Proceedings of the Symposium on Applied Computing*, vol. Part F128005, pp. 1280–1287, 4 2017. [Online]. Available: https://dl.acm.org/doi/10.1145/3019612.3019788

[6] Y. Jia *et al.*, "Ekd-bsp: Bug report severity prediction by extracting keywords from description," *2021 8th International Conference on Dependable Systems and Their Applications (DSA)*, pp. 42–53, 8 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9623014/

[7] E. Mashhadi, H. Ahmadvand, and H. Hemmati, "Method-level bug severity prediction using source code metrics and llms," in *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. Los Alamitos, CA, USA: IEEE Computer Society, oct 2023, pp. 635–646. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ISSRE59848.2023.00055

[8] G. Yang, T. Zhang, and B. Lee, "Towards semi-automatic bug triage and severity prediction based on topic model and multi-feature of bug reports," *2014 IEEE 38th Annual Computer Software and Applications Conference*, pp. 97–106, 7 2014. [Online]. Available: https://ieeexplore.ieee.org/document/6899206

[9] Y. Su *et al.*, "Cil-bsp: Bug report severity prediction based on class imbalanced learning," *2022 IEEE 22nd International Conference on Software Quality, Reliability, and Security Companion (QRS-C)*, pp. 298–306, 12 2022. [Online]. Available: https://ieeexplore.ieee.org/document/10076993/

[10] G. Ke *et al.*, "Lightgbm: A highly efficient gradient boosting decision tree," in *Advances in Neural Information Processing Systems*, I. Guyon *et al.*, Eds., vol. 30. Curran Associates, Inc., 2017, pp. 3149–3157. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf

[11] A.-H. Dao, Y.-L. Yu, and C.-Z. Yang, "Severity prediction for bug reports using tree-based ensemble models: A comparative study," *2022 IEEE 2nd International Conference on Software Engineering and Artificial Intelligence (SEAI)*, pp. 01–05, 6 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9832212/

[12] R. M. D. S. Rathnayake, B. T. G. S. Kumara, and E. M. U. W. J. B. Ekanayake, "Cnn based severity prediction of bug reports," *2021 From Innovation To Impact (FITI)*, pp. 1–6, 12 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9833043/

[13] W. Y. Ramay *et al.*, "Deep neural network-based severity prediction of bug reports," *IEEE Access*, vol. 7, pp. 46 846–46 857, 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8685106/

[14] J. Kim and G. Yang, "Bug severity prediction algorithm using topic-based feature selection and cnn-lstm algorithm," *IEEE Access*, vol. 10, pp. 94 643–94 651, 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9878105/

[15] N. K.-S. Roy and B. Rossi, "Towards an improvement of bug severity classification," *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, pp. 269–276, 8 2014. [Online]. Available: https://ieeexplore.ieee.org/document/6928822/

[16] Y. Tian, D. Lo, and C. Sun, "Information retrieval based nearest neighbor classification for fine-grained bug severity prediction," *2012 19th Working Conference on Reverse Engineering*, pp. 215–224, 10 2012. [Online]. Available: http://ieeexplore.ieee.org/document/6385117/

[17] R. Hazra, A. Dwivedi, and A. Mukherjee, "Is this bug severe? a text-cum-graph based model for bug severity prediction," in *Machine Learning and Knowledge Discovery in Databases. ECML PKDD 2022*, ser. Lecture Notes in Computer Science, M. R. Amini

et al., Eds. Springer, Cham, 2023, vol. 13718, pp. 236–252, published 18 March 2023. [Online]. Available: https://doi.org/10.1007/978-3-031-26422-1_15

[18] J. C. Campbell, E. A. Santos, and A. Hindle, "The unreasonable effectiveness of traditional information retrieval in crash report deduplication," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 269–280. [Online]. Available: https://doi.org/10.1145/2901739.2901766

[19] I. M. Rodrigues, "Algorithms and learning models for bug report deduplication," Ph.D. dissertation, Polytechnique Montréal, May 2022. [Online]. Available: https://publications.polymtl.ca/10297/

[20] H. Touvron *et al.*, "Llama 2: Open foundation and fine-tuned chat models," 7 2023. [Online]. Available: http://arxiv.org/abs/2307.09288

[21] T. Brown *et al.*, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, H. Larochelle *et al.*, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[22] A. Chowdhery *et al.*, "Palm: scaling language modeling with pathways," *Journal of Machine Learning Research*, vol. 24, no. 1, mar 2024.

[23] J. Hoffmann *et al.*, "Training compute-optimal large language models," 3 2022. [Online]. Available: http://arxiv.org/abs/2203.15556

[24] H. Touvron *et al.*, "Llama: Open and efficient foundation language models," 2 2023. [Online]. Available: http://arxiv.org/abs/2302.13971

[25] U. Kamath, J. Liu, and J. Whitaker, *Deep Learning for NLP and Speech Recognition*. Springer International Publishing, 1 2019. [Online]. Available: http://link.springer.com/10.1007/978-3-030-14596-5

[26] E. F. Tjong Kim Sang and F. De Meulder, "Introduction to the CoNLL-2003 shared task: Language-independent named entity recognition," in *Proceedings of the Seventh Conference on Natural Language Learning at HLT-NAACL 2003*, 2003, pp. 142–147. [Online]. Available: https://aclanthology.org/W03-0419

[27] M. Banko and R. C. Moore, "Part-of-speech tagging in context," in *COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics*.

Geneva, Switzerland: COLING, aug 23–aug 27 2004, pp. 556–561. [Online]. Available: https://aclanthology.org/C04-1080

[28] T. Mikolov *et al.*, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 1 2013. [Online]. Available: http://arxiv.org/abs/1301.3781

[29] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proceedings of the 31st International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, E. P. Xing and T. Jebara, Eds., vol. 32, no. 2. Bejing, China: PMLR, 22–24 Jun 2014, pp. 1188–1196. [Online]. Available: https://proceedings.mlr.press/v32/le14.html

[30] D. Cer *et al.*, "Universal sentence encoder," *CoRR*, vol. abs/1803.11175, 2018. [Online]. Available: http://arxiv.org/abs/1803.11175

[31] S. Wold, K. Esbensen, and P. Geladi, "Principal component analysis," *Chemometrics and Intelligent Laboratory Systems*, vol. 2, no. 1, pp. 37–52, 1987, proceedings of the Multivariate Statistical Workshop for Geologists and Geochemists. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0169743987800849

[32] C. F. Van Loan, "Generalizing the singular value decomposition," *SIAM Journal on Numerical Analysis*, vol. 13, no. 1, pp. 76–83, 1976. [Online]. Available: https://doi.org/10.1137/0713009

[33] J. Devlin *et al.*, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. [Online]. Available: https://aclanthology.org/N19-1423

[34] A. Radford *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019. [Online]. Available: https://d4mucfpksywv.cloudfront.net/better-language-models/language_models_are_unsupervised_multitask_learners.pdf

[35] OpenAI *et al.*, "Gpt-4 technical report," 3 2023. [Online]. Available: http://arxiv.org/abs/2303.08774

[36] Y. Wei, C. Zhang, and T. Ren, "Improving bug severity prediction with domain-specific representation learning," *IEEE Access*, vol. 11, pp. 62 829–62 839, 2023. [Online]. Available: https://ieeexplore.ieee.org/document/10131903/

[37] Y. Liu *et al.*, "Roberta: A robustly optimized BERT pretraining approach," *CoRR*, vol. abs/1907.11692, 2019. [Online]. Available: http://arxiv.org/abs/1907.11692

[38] M. Lewis *et al.*, "Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," *CoRR*, vol. abs/1910.13461, 10 2019. [Online]. Available: http://arxiv.org/abs/1910.13461

[39] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, K. Erk and N. A. Smith, Eds. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 1715–1725. [Online]. Available: https://aclanthology.org/P16-1162

[40] Hugging Face, "Hugging Face NLP Course," https://huggingface.co/learn/nlp-course/, 2024, accessed: January 29, 2024.

[41] H. Lin *et al.*, "Towards explainable harmful meme detection through multimodal debate between large language models," 1 2024. [Online]. Available: http://arxiv.org/abs/2401.13298

[42] J. Pang *et al.*, "Salute the classic: Revisiting challenges of machine translation in the age of large language models," 1 2024. [Online]. Available: http://arxiv.org/abs/2401.08350

[43] P. F. Simmering and P. Huoviala, "Large language models for aspect-based sentiment analysis," 10 2023. [Online]. Available: http://arxiv.org/abs/2310.18025

[44] P. Přibáň *et al.*, "A comparative study of cross-lingual sentiment analysis," *Expert Systems with Applications*, vol. 247, p. 123247, 8 2024. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S095741742400112X

[45] L. Zhang *et al.*, "Deep de finetti: Recovering topic distributions from large language models," 12 2023. [Online]. Available: http://arxiv.org/abs/2312.14226

[46] Y. Kim *et al.*, "Health-llm: Large language models for health prediction via wearable sensor data," 1 2024. [Online]. Available: http://arxiv.org/abs/2401.06866

[47] T. Li *et al.*, "Cancergpt for few shot drug pair synergy prediction using large pretrained language models," *npj Digital Medicine*, vol. 7, p. 40, 2 2024. [Online]. Available: https://www.nature.com/articles/s41746-024-01024-9

[48] Y. Liu *et al.*, "Investigating the fairness of large language models for predictions on tabular data," 10 2023. [Online]. Available: http://arxiv.org/abs/2310.14607

[49] M. Fonseca and S. B. Cohen, "Can large language model summarizers adapt to diverse scientific communication goals?" 1 2024. [Online]. Available: http://arxiv.org/abs/2401.10415

[50] T. Dettmers *et al.*, "Advances in neural information processing systems 36 - 37th conference on neural information processing systems, neurips 2023," in *Proceedings of the 37th Conference on Neural Information Processing Systems (NeurIPS)*, no. 1, NeurIPS. New Orleans, LA, USA: NeurIPS, 2023.

[51] E. J. Hu *et al.*, "Lora: Low-rank adaptation of large language models," 6 2021. [Online]. Available: http://arxiv.org/abs/2106.09685

[52] Bugzilla Project, "Bugzilla Documentation," https://bugzilla.readthedocs.io/en/5.2/index.html, accessed on: February 13, 2024.

[53] A.-H. Dao and C.-Z. Yang, "Severity prediction for bug reports using multi-aspect features: A deep learning approach," *Mathematics*, vol. 9, p. 1644, 7 2021. [Online]. Available: https://www.mdpi.com/2227-7390/9/14/1644

[54] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," *2008 IEEE International Conference on Software Maintenance*, pp. 346–355, 9 2008. [Online]. Available: http://ieeexplore.ieee.org/document/4658083/

[55] W. W. Cohen, "Fast effective rule induction," in *Machine Learning Proceedings 1995*, A. Prieditis and S. Russell, Eds. San Francisco (CA): Morgan Kaufmann, 1995, pp. 115–123. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9781558603776500232

[56] R. Mihalcea and P. Tarau, "Textrank: Bringing order into text," in *Conference on Empirical Methods in Natural Language Processing*, 2004. [Online]. Available: https://api.semanticscholar.org/CorpusID:577937

[57] A. Joulin *et al.*, "Bag of tricks for efficient text classification," *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers*, vol. 2, pp. 427–431, 2017. [Online]. Available: http://aclweb.org/anthology/E17-2068

[58] N. V. Chawla *et al.*, "Smote: Synthetic minority over-sampling technique," *Journal of Artificial Intelligence Research*, vol. 16, pp. 321–357, 6 2011. [Online]. Available: http://arxiv.org/abs/1106.1813http://dx.doi.org/10.1613/jair.953

[59] L. Breiman, "Random forests," pp. 5–32, 10 2001. [Online]. Available: https://doi.org/10.1023/A:1010933404324

[60] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Machine Learning*, vol. 63, pp. 3–42, 4 2006. [Online]. Available: http://link.springer.com/10.1007/s10994-006-6226-1

[61] T. Chen and C. Guestrin, "Xgboost," *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 785–794, 8 2016. [Online]. Available: https://dl.acm.org/doi/10.1145/2939672.2939785

[62] P. Goyal, S. Pandey, and K. Jain, *Deep Learning for Natural Language Processing*. Apress, 2018. [Online]. Available: http://link.springer.com/10.1007/978-1-4842-3685-7

[63] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing*. Association for Computational Linguistics, 11 2019, pp. 3982–3992. [Online]. Available: https://aclanthology.org/D19-1410.pdf

[64] P. Veličković *et al.*, "Graph Attention Networks," *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=rJXMpikCZ

[65] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. [Online]. Available: https://openreview.net/forum?id=SJU4ayYgl

[66] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in Neural Information Processing Systems*, I. Guyon *et al.*, Eds., vol. 30. Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/file/5dd9db5e033da9c6fb5ba83c7a7ebea9-Paper.pdf

[67] Z. Feng *et al.*, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, T. Cohn, Y. He, and Y. Liu, Eds. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.139

[68] Y. Tan *et al.*, "Bug severity prediction using question-and-answer pairs from stack overflow," *Journal of Systems and Software*, vol. 165, p. 110567, 7 2020. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0164121220300480

[69] S. Fang *et al.*, "Effective prediction of bug-fixing priority via weighted graph convolutional networks," *IEEE Transactions on Reliability*, vol. 70, pp. 563–574, 6 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9435622/

[70] A. Kukkar *et al.*, "A novel deep-learning-based bug severity classification technique using convolutional neural networks and random forest with boosting," *Sensors*, vol. 19, p. 2964, 7 2019. [Online]. Available: https://www.mdpi.com/1424-8220/19/13/2964

APPENDIX A PROMPT TEMPLATES

In this appendix, we present the two prompt templates used in the input of Large Language Models. We adopt the *Official* template provided by Meta and an alternative template called *Alpaca*, which is a commonly used prompt template with LLMs. These templates follow a structured format:

1. Official Template:

```
<s>[INST] <<SYS>>
{your_system_message}
<</SYS>>
{user_message_1} [/INST]
```

2. Alpaca Template:

```
{pre prompt}

### Instruction:
{instruction}

### Input:
{input}

### Response:
```

In the Offical template, `your_system_message` variable provides instructions for the system, directing it to respond with a single token and categorize the bug report as either "0 = NOT SEVERE" or "1 = SEVERE." The instruction string emphasizes brevity and specificity, guiding the system to focus solely on the severity categorization task. In the Alpaca template, these instructions are placed in the `pre prompt` and in the variable `instruction`.

The `user_message_1` variable contains the bug report's description and a reminder of the categorization instructions. In the Alpaca template, the description of the bug report is placed in the variable `input`, and we add a reminder of the task along with the `###` symbols. This ensures that the bug report is accurately categorized based on the provided guidelines.

The final template formats are as follows:

1. Official Template:

```
[INST] <<SYS>>
Always answer with one token. Do not give any explanation. Use
only 0 or 1 and one token. Skip any politeness answer. You have
only one word available.
<</SYS>>
Categorize the bug report into one of 2 categories:

0 = NOT SEVERE
1 = SEVERE

Bug report:
$bug_description

Remembering the instruction:
Categorize the bug report into one of 2 categories:

0 = NOT SEVERE
1 = SEVERE[/INST]
```

2. Alpaca Template:

```
Always answer with one token. Do not give any explanation. Use
only 0 or 1 and one token. Skip any politeness answer. You have
only one word available.
Below is an instruction that describes a task. Write a response
that appropriately completes the request.

### Instruction:
Categorize the bug report into one of 2 categories:

0 = NOT SEVERE
1 = SEVERE

### Input:
$bug_description

### Remembering the instruction:
Categorize the bug report into one of 2 categories:

0 = NOT SEVERE
1 = SEVERE

### Response:
```

Here, `$bug_description` represents the description field of the bug report, ensuring that all relevant information is included in the input prompt. This standardized template facilitates effective communication between the user and the LLM model, streamlining the bug severity prediction process.