UNIVERSIDADE
BEIRA INTERIOR

Bug Taxonomy Classification Using Machine Learning Algorithms

Beatriz Isabel Trocas Caldeira

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática
(2º ciclo de estudos)

Orientador: Prof. Doutor Nuno Gonçalo Coelho Costa Pombo

Covilhã, setembro de 2024

Declaração de Integridade

Eu, Beatriz Isabel Trocas Caldeira, que abaixo assino, estudante com o número de inscrição M12432 do Mestrado em Engenharia Informática da Faculdade Engenharia, declaro ter desenvolvido o presente trabalho e elaborado o presente texto em total consonância com o Código de Integridades da Universidade da Beira Interior.

Mais concretamente afirmo não ter incorrido em qualquer das variedades de Fraude Académica, e que aqui declaro conhecer, que em particular atendi à exigida referenciação de frases, extratos, imagens e outras formas de trabalho intelectual, e assumindo assim na íntegra as responsabilidades da autoria.

Universidade da Beira Interior, Covilhã 27/9/2024

Acknowledgements

I want to start by thanking my family for everything they have done for me, specially the sacrifices you have made that allowed me to have the privilege and opportunity to continue to pursue my academic career this far. I hope I made it all worth it and I am forever grateful for everything. Individually, to my mother, who is the most resilient and versatile woman that can turn dust into gold and who has endured so much but always stayed with her chin up and her arms open to give a hug whenever needed. I am eternally grateful for your love and patience. To my father, a restless hard-working man of many traits who has always encouraged me to never settle for less and to always push myself forward. To my sister, who always stands alongside me and encourages me to be the best version of myself. It is a pleasure to grow up and experience life with you, no matter how far we are from each other. Thank you all for teaching me work ethics, essential for completing my academic degree, and for teaching me important life lessons that I will carry with me forever. I could not have done it without all your support, company and guidance.

Furthermore, I would like to thank my advisor, Professor Nuno Pombo, for the opportunity to work together and for all the knowledge transmitted as well as the patience to guide me throughout this long journey. Moreover, I want to acknowledge the presence and support of my friends, for putting up with me when things go wrong, for getting me out of the house, for the guitar gigs that warmed my heart, and for sharing your laughter and life with me. Last but not least, I want to thank the city of Covilhã, for being my second home for 5 years, and for the prettiest sunrises that lit my day when times were hard.

I hope that I have met your expectations and that my efforts made all of you proud. The presentation of this body of work marks the conclusion of my academic journey and the attainment of a significant personal achievement I considered impossible for me for most of my life. As I embark in new challenges, opportunities and ”eras” that will come my way, I look forward to having the pleasure of benefiting from your support and company.

I am eternally grateful.

Resumo

Bugs são uma ocorrência natural no mundo do desenvolvimento de software. Numa sociedade com cada vez mais software construído, o número destas ocorrências também tem vindo a aumentar, e podem ter consequências catastróficas para um produto ou negócio. De forma a tentar monitorizar e acompanhar o processo de resolução de bugs, foram criadas plataformas e sistemas de rastreamento dos mesmos. Através destas plataformas, as equipas responsabilizadas pelo processo de análise de bugs podem ver quais são os bugs que foram identificados, quais são os mais comumns e em que fase de resolução estes se encontram. O problema depara-se com a análise de cada um destes relatórios. Em produtos de grande dimensão centenas de relatórios podem ser submetidos nas suas plataformas por dia, quer por outros desenvolvedores, pen-testers ou por utilizadores finais. Os utilizadores finais, devido à sua provável falta de conhecimento acerca de software, da arquitetura do sistema que usam, ou de outros processos inerentes, podem submeter relatórios errados ou pouco claros. Combater a necessidade de desenvolvedores analisarem e categorizarem cada um destes relatórios (que é bastante complexo e demoroso), através de automatização é uma ideia que tem vindo a ser estudada por vários investigadores.

Esta investigação visa a construir um esquema de classificação o mais abrangente e detalhado possível que possa fornecer informação extra e automatizada a um analisador de relatórios, para que uma parte da sua análise possa ser facilitada. Como tal, a solução encontrada basou-se na criação de uma implementação que tira partido da elevada performance e capacidades do modelo Bidirectional Encoder Representations from Transformers (BERT), um modelo baseado na arquitetura Transformers, para classificar estes relatórios da forma mais precisa possível, baseando-se nas descrições textuais presentes em cada um. Para isto foi criado um dataset baseado nesse esquema, desenhado através de uma solução divide-and-conquer e vários modelos foram treinados em cada uma das categorias. Os resultados mostram que o BERT pode fazer parte de uma solução bastante robusta para este propósito, mesmo apesar da pequena quantidade de dados etiquetados que lhe foram dados como input e da necessidade de algum tipo de refinamento do processo de treino. Os melhores resultados foram obtidos quando o titulo e a descrição eram usados em conjunto como dado de entrada para o modelo, um deles conseguindo até precisão geral de 75%, sendo a mais fraca de 54.6%.

Palavras-chave

BERT, Relatórios de Bugs, Sistemas de Rastreamento de Bugs, Classificação de Bugs, Triagem de Bugs, Aprendizagem Supervisionada, Processamento de Linguagem Natural, Mineração de Dados, Aprendizagem de Máquina, Modelos de Linguagem de Grande Escala.

Abstract

Bugs are a natural occurrence in the realm of software development. As society increasingly relies on software, the frequency of these occurrences has naturally increased as well, potentially leading to catastrophic consequences for products or businesses. To monitor and manage the bug resolution process, various bug tracking systems have been developed. These platforms enable teams responsible for bug analysis to view identified bugs, track the most common issues, and monitor their resolution status. However, the challenge lies in analyzing each of these reports. In large-scale products, hundreds of reports can be submitted daily through these platforms, whether by other developers, pen testers, or end users. End users, due to their likely lack of knowledge about software, system architecture, or other inherent processes, may submit incorrect or unclear reports. Addressing the need for developers to manually analyze and classify each of these reports (a complex and time-consuming task) through automation is an idea that has been explored by various researchers.

This research aims to develop a comprehensive and detailed classification schema that can provide additional, automated information to report analysts, thereby facilitating part of their analysis process. To achieve this, the proposed solution is based on leveraging the high performance and capabilities of the BERT model, a model rooted on Transformer architecture, to classify these reports as accurately as possible, based on the textual descriptions within each report. A dataset was created following this schema, designed through a divide-and-conquer approach, and multiple models were trained on each category. The results indicate that BERT can form the basis of a robust solution for this purpose, even when provided with a small amount of labeled data as input and despite the need for some refinement in the training process. The best results were obtained when both the title and the description were used together as input data for the model, with one model achieving an overall accuracy of 75%, and the lowest accuracy being 54.6%.

Keywords

BERT, Bug Reports, Bug Tracking Systems, Bug Classification, Bug Triage, Supervised Learning, Natural Language Processing, Data Mining, Machine Learning, Large Language Models.

Contents

List of Figures

List of Tables

List of Acronyms

API Application Programming Interface

AUC-ROC Area Under ROC Curve

BERT Bidirectional Encoder Representations from Transformers

BOW Bag of Words

BR Bug Report

BTS Bug Tracking System(s)

DBRNN-A Attention-based Deep Bidirectional Recurrent Neural Networks

GUI Graphic User Interface

GPT Generative Pre-training Transformer

GPU Graphics Processing Unit

LDA Latent Dirichlet Allocation

LiDA Linear Discriminant Analysis

LLM Large Language Models

ML Machine Learning

NB Naïve Bayes

NLP Natural Language Processing

ODC Orthogonal Defect Classification

PCA Principal Component Analysis

SVM Support Vector Machines

TF Term Frequency

TF-IDF Term Frequency - Inverse Document Frequency

UBI Universidade da Beira Interior

URL Uniform Resource Locator

UI User Interface

Chapter 1

Introduction

1.1 Background

Software errors, also commonly known as bugs and software defects, are defined as an incorrect step, process, or data definition in a computer program, causing the said program to output an incorrect result [8], meaning that it does not work as it was designed / planned to function. These are quite common, as the industry average is 1-25 errors per 1,000 lines of code for delivered software [9]. This average depends on the development life cycle, practices, and other processes. This information implies that a small application of 20,000 lines of code is likely to have around 200 serious coding errors. Although some of these errors come from bad design or requirements, in large projects, some researchers have reported that about 75% of these errors come from software construction / coding [10]. Identifying these errors mainly through software testing is essential, and tracking them ensures that they are fixed.

Bug Tracking System(s) (BTS) are tools that allow bugs in software to be reported and tracked / monitored efficiently. These tools are essential and commonly used in software development projects, especially when this software is in its testing phase. It also allows developers to classify the prioritization and importance of a certain issue / bug, which helps debugging teams manage their work and tasks more efficiently. All of this is even more important when the system under evaluation is large and complex in order to meet diversified user requirements and, as such, might be more likely to have a larger number of issues [11].

Although software bugs can be considered as software-only and technical problems, they can influence the business as a whole, including management decisions. As Robert B. Grady mentioned, ”(...) software defect data is your most important available management information source for software process improvement decisions (...). Ignoring defect data can lead to serious consequences for an organization’s business.” [3]. These consequences include expensive updates after the product has been released (to repair defects) and a poor quality reputation as a result [3]. This highlights the importance of BTS and how valuable they can be to the business that uses them. Some of the benefits of using these types of tools include facilitating the prioritization of bugs, helping in the management of bugs for future releases, increasing transparency of the development process, aiding the planning of new releases, and increasing the detection of application-based bugs [11].

Some BTS, such as Bugzilla, GitHub, and Atlassian’s page on JIRA, are also available to the public instead of only development or quality assurance teams and are very common, especially in open source projects because the developers are spread across the world and individuals can participate in the process. As mentioned above, users can report issues they have had to the platform through a text description of the problem and the steps to reproduce

it, and then each bug report is eventually evaluated by a developer. Although this possibility is great because developers can now know the main issues most of their clients are facing, it also has some downsides, such as repeated issues (which is a huge problem in itself), inaccurate Bug Report (BR), and misclassifications. Some platforms allow the user to classify the issue into certain classes (predefined by the developers), but since the user's knowledge on software development might be limited, the issue might end up misclassified. This lack of knowledge can be very detrimental to a project, as, for example, security-related bugs can end up being classified as non-security related and thus having a lower fix priority for debugging teams, which ultimately ends up leaving the software open to security breaches that can be exploited by attackers [12]. The end-user also tends to have a different insight on what might be considered a bug, compared to a developer, because users have no project insight or understanding of its technical details, which leads to decreased quality of the bug tracking side of a project. A study showed (through 90 work-days of manual analysis of a bug report dataset) that 33.8% of bug reports refer to the possible implementation of new features, updates in the supporting documentation, or internal refactoring [13]. This percentage shows how much bias is likely to be included in bug-type prediction models and how it affects the end result of the models developed. Currently, evaluating this user classification, the complete bug report, the issue itself, and organizing the bugs is very time-consuming, repetitive, and tedious because it is done manually and, as such, can be error prone [14]. This process (shown in full in Figure 1.1) can change in the near future.

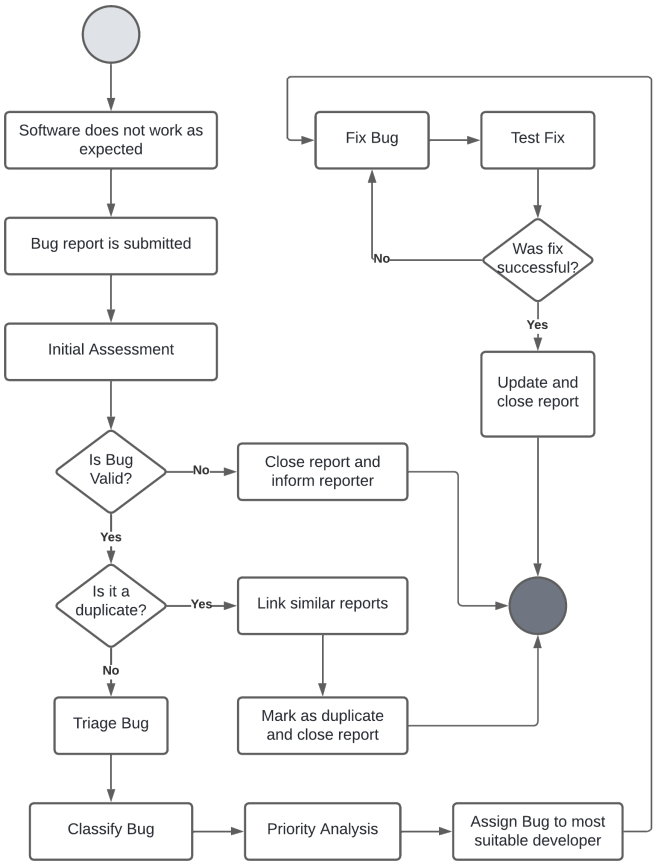


Figure 1.1: Flowchart of the software bug report classification process, adapted from [1].

Classifying issues through Machine Learning (ML) models allows teams to dedicate more time to solving the software problem at hand, which may allow shorter bug resolution times and eventually lead to better software as a whole. These automatic classifications can also be used to help train ML models on assigning bugs to the most relevant developer in the issue's field (bug triage), removing an additional layer of manual work for developers.

1.2 Objective and Scope

The objectives of this dissertation are the following:

1. Create a versatile classification of bug taxonomies, abstract enough to be used in a large percentage of software developed, despite its architecture. This will be done by studying the strengths and weaknesses of the studies mentioned in the literature review;
2. Create a machine learning approach whilst using bug descriptions from true public bug reports for automatic classification in certain taxonomies, mentioned in the previous objective. The performance of the model shall then be compared to the studies presented in the literature review;

1.3 Structure Overview

This dissertation is composed of four main chapters, each of which can be summarized as follows:

- **Chapter One - Introduction:** Presents the problem at hand, providing some insights into its consequences, as well as the objective in sight, and descriptions of the document's organization;
- **Chapter Two - Literature Review:** Presents the techniques applied in this field by other researchers in their implementations, and respective results.
- **Chapter Three - Methodology:** Presents an overview of the work outlined for this dissertation, along with insights and descriptions of the concepts and its context related to the implementation of the various techniques;
- **Chapter Four - Results:** Presents the results achieved through the methodology, and their description;
- **Chapter Five - Conclusions:** Discusses the conclusions drawn from the preparation of the present document, and what could be added to this implementation to get enhanced results.

Chapter 2

Literature Review

2.1 Introduction

In the pursuit of knowledge and innovation, every researcher ”stands on the shoulders of giants” as they try to build upon the foundations of work created by those who came before them. These foundations steer emerging researchers toward new paths full of potential, while barring entry to paths that were shown to be unsuccessful. As such, this chapter serves as an exploration of the existing body of research, providing a panoramic view of the current state of the art in automatic bug taxonomy classification.

The last two decades have witnessed a substantial *boom* in software development and, with it, so have studies on BR and BTS. Most studies and approaches in this field try to find methodologies to identify the developer most relevant to fix a bug (bug triage), classify bugs as duplicates, classify bugs as security / non-security related, and classify if a BR is truly a bug or not. Although all of these studies are extremely useful, there are few approaches to support developers in the first analysis of the BR, even though this initial step is the most time consuming in the bug triage process [7]. As such, although some of the studies analyzed throughout this dissertation do not have the same exact objectives as this one, they were still important to guide this investigation on what the most common practices were for these types of problems.

2.2 Bug Classification Schemes

Since software exists, its defects needed to be classified into different categories, for quality assurance purposes. As such, the researchers tried to formulate diverse classification frameworks that could fit the most relevant bug types. However, the dynamic evolution of software design, technological advancements, and the spectrum of problems it fixes have established some variations. These classification schemes will continue to go through adaptation and refinement, as software is ever-changing, although many are still trying to remain true to the foundation concepts of the earlier studies, which remain relevant.

In a 2005 publication, Freimut et al. [15] presented a set of attributes and criteria that could be used to examine the quality of a defect categorization scheme. The researchers affirmed that a scheme is considered good if: it presents **clear and meaningful definitions** for all categories; it **defines** the values of the attributes in each category; different developers give the same classifications with a high degree of agreement (**reliability**); it ensures that every defect is classifiable using the built scheme; it has **5 to 9 categories** (number of items that a human short-term memory can retain [16] [17]); it has **examples** given for

each category. This criteria is important to keep in mind, to better understand the possible advantages and downsides for the following studies.

Numerous defect classification frameworks have been established drawing inspiration from the Orthogonal Defect Classification (ODC) concept introduced in 1992 by Chillarege and Bhandari et al. [16]. This concept was developed with the primary objective of extracting information about the software development process from detected defects. Consequently, within the ODC framework, the authors mention that the categorizations of defect types should exhibit clarity and transparency, ensuring that programmers can clearly understand them, without the risk of ambiguity or confusion [16]. These pertinent categories comprise the following:

- **Function error:** affects capability, end-user interfaces, product interfaces and require a formal design change;
- **Assignment error:** errors in few lines of code, such as the initialization of control blocks or data structure;
- **Interface:** errors in interacting with other components, call statements, control blocks and others;
- **Checking:** problems with logic in data validation;
- **Timing:** errors in shared and real-time resources;
- **Build / package / merge:** errors that occur due to mistakes in libraries, version control or management changes;
- **Algorithm:** efficiency problems that can be fixed by re-implementing an algorithm or data structure without a design change.

The researchers derived these concepts through a rigorous analysis of both the symptoms of the defects and their underlying causes. As such, ODC can indicate the structure of the software involved in the failure but not the type of issue, and some researchers argue that this classification scheme should be used in a complementary manner with other taxonomies to better understand the issue at hand. An example given by Catolino et al. mentions that ”a developer can first use our taxonomy to understand the type of bug that occurred and then refine the process by using ODC to characterize the program structure causing that bug type” [7].

Continuing the exploration of the contributions of Catolino et al., these researchers also studied the field of automatic bug classification, further mentioned in 2.3, and created a method in which manual labeling was needed [7]. As such, the bug reports were classified as belonging to one of the following relevant categories:

- **Configuration:** caused by wrong usage of external dependencies or issues in configuration files (*xml* or *manifest* artifacts);
- **Network:** server issues due to a network problem, unexpected server shutdowns, or misused communication protocols;

- **Database:** problems in queries or database connection;
- **GUI:** styling errors and unexpected failures in the form of unusual error messages;
- **Performance:** memory overuse, endless loops and energy leaks;
- **Permission / deprecation:** related to presence, modification or removal of deprecated methods or Application Programming Interface (API)s (and their permissions);
- **Security:** vulnerabilities that commonly refer to unused permissions and reloading parameters;
- **Program anomaly:** wrong return values, exceptions, unexpected crashes and other issues related to code logic.

In a similar approach, Zhang et al. [18], also mentioned in 2.3, needed to come up with a classification scheme to label their data as well. Some of the types defined include the following:

- **Content:** containing issues of subtypes related to Static Content, Data Transmission, Data access, and Data management;
- **User Interface:** containing issues of subtypes issues related to Layout, Compatibility, Links, Readability, Accessibility, Cookie, Streaming Content and Client-side Scripting;
- **Navigation:** containing issues of subtypes related to Bookmarks, Redirect, Frames and Framesets, Site Maps and Internal Search Engines;
- **Component:** issues related to Style, Validation, Storage, and Exception;
- **Security:** with no related subtypes;
- **Configuration:** with no related subtypes;
- **Performance:** with no related subtypes.

It is important to emphasize that the category scheme created in this study and its labels are specifically designed for use in web-based applications, which accounts for their strong reliance on bugs related almost exclusively to visual elements. This focus on the visual aspect explains the limited and almost non-existing differentiation among backend categories. However, given the streamlined nature of this study, these classifications can serve as a foundation for researchers to develop more comprehensive frameworks that incorporate a broader range of backend categories.

The Hewlett-Packard Software Metrics Council created a standard defect terminology that could be used for multiple different HP projects [3]. The classification model uses the following image as definitions:

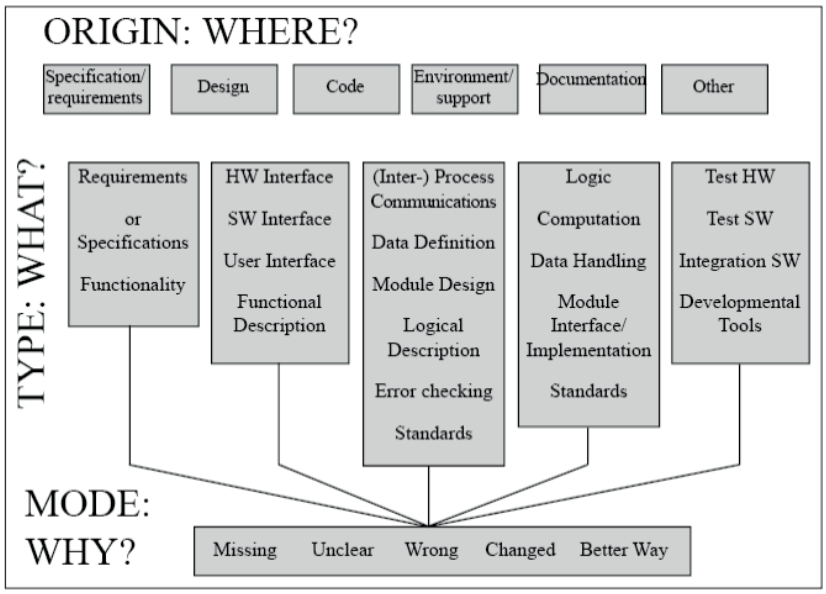


Figure 2.1: Categorization of Software Defects used by Hewlett-Packard (1996) [2][3].

These types of definitions are meant to be used by selecting one descriptor for each one of the questions (Where? What? Why?). As an illustration, consider a situation where there is a defect, such as a ”Design” flaw, where a portion of the ”User Interface” outlined in the specification is ”Missing”. Despite being created nearly 30 years ago, the categories in this scheme remain highly relevant and applicable to contemporary software issues. This enduring relevance is one of the key reasons why the study continues to be widely cited in the field.

Endres, from IBM Laboratory in Germany [19], was among the early trailblazers in the field. In 1975, he authored a paper on the analysis of error causes in system programs. In this research, Endres introduced an error protocol form that, during its initial phase, classified problems into various generalized groups, such as: Machine Error; User or operator error; Suggestion for improvement; Documentation Error and finally Program error (not previously identified). The researcher then created a great number of complex subgroups made of types of errors, such as:

- Errors in the understanding of the problem and choice of algorithm to solve it;
- Errors that lie in the implementation of the algorithm, such as translation of an algorithm into a programming language;
- Errors that the author describes as ”errors in the code which must not remain there (...) and can be removed by people who are not programmers or have no detailed knowledge of the project” [19].

Each of these subgroups was then divided into more sections, which give more detailed (although extensive) information about the type of bug, but which do not align with the objective of this dissertation.

In 1982, Ostrand et al. mentioned that a number of studies and issue classification schemes created to date included ”ambiguous, overlapping, and incomplete categories, too

many categories, and confusion of error symptoms, error causes, and actual errors" [20]. Although the scheme created by this study is mainly focused on the programming aspect of errors, it still remains a staple in this field many decades later, as it laid the groundwork for numerous subsequent research endeavors in the field, especially given its publication date. The relevant attributes are the following (as written by the authors):

- **Major Category:**
  - Data definition - Code which defines constants, storage areas, control codes, etc;
  - Data Handling - Code which modifies or initializes the values of variables;
  - System - Error in the program's environment, including operating System, compiler, hardware, etc;
- **Type:**
  - Address - Information which locates values in memory (array index, list pointer, and others);
  - Control - (...) code identifying different output display formats;
  - Data - Primary information which is processed, read, or written;
- **Presence:**
  - Omitted - Something was left out;
  - Superfluous - Something was present that should not have been;
  - Incorrect - Something present had to be corrected.

In a real-life context, an issue described as "When output screen C was re-displayed to add more names to the destination-table, source lines were created with the 'indexed-by' phrase overlaying the 'destination-table' phrase" would be classified as Data Definition / Control / Omit [20].

While in the mission of aggregating historical datasets of defect data using different categorization schemes, Seaman and Shull et al. [21] created a scheme based on ODC [16], to facilitate the mapping of issues found in NASA software inspections. The created scheme involves the following categories on design and source code inspection defect types (as described by the authors) [21]:

- **Algorithm / method:** Error in the set of steps used to solve a problem or computation, including incorrect implementation of algorithms, or calls to inappropriate functions;
- **Assignment / initialization:** A variable that is assigned a value incorrectly or is not initialized properly;
- **Checking:** Inadequate checking for potential error conditions;
- **Data:** Incorrectly defined data structure, pointer or memory allocation errors;

- **External interface:** Errors in the User Interface (UI) (including usability problems);
- **Logic:** Incorrect logical conditions on if, case or loop blocks;
- **Non-functional defects:** Failure to meet non-functional requirements such as performance, and lack of clarity of the design or code to the reader (comments included);
- **Timing / Optimization:** Errors that will cause timing or performance problems (e.g. unnecessarily slow implementation of an algorithm).

These categories were delineated through a meticulous examination of NASA software as opposed to encompassing general or open-source software, yet a substantial proportion of these categories remain important for consideration within the context of this research. The authors also presented a challenge encountered during the design of the categories that refers to the issue of orthogonality. Given that these categories lacked mutual exclusivity, certain defects posed difficulty in consistent classification. This challenge stemmed from the fact that these defects had to be assigned only to one specific category, although they could be assigned to multiple categories concurrently.

Sullivan and Chillarege (1992) [22] at IBM defined the difference between what is an error type and a defect type. According to the researchers, a defect type is a higher-level classification that includes and distinguishes between UI / design mistakes, coding errors, timing errors, and administrative mistakes, while an error type is a more low-level programming mistake that led to software failure. As such, the researchers classified the type of Software Defect as a function defect, algorithm defect, assignment defect, interface defect, timing defect, and build / merge defect (due to version control errors).

More recently, in 2023, some researchers proposed a new classification scheme, based on the work of Sullivan et al. [22] and others. The proposed Defect Classification Scheme by Sultan et al. include the following relevant types [23]:

- **Interface errors:**Errors that visually appear in the UI;
- **Calculation errors:** Errors that appear in areas that had calculations done before;
- **Loading errors:** Errors in response time for loading data;
- **Security errors:** Errors in the imbalance of privileges and rules for each user;
- **Business Logic errors:** Inconsistencies within the logic.

This scheme provides a good general description of the types of defects that exist, although it is not very detailed because no subtypes were defined.

Considering the publication dates of certain previously mentioned studies, it is imperative to recognize that, while they continue to hold contemporary relevance within the realm of current software development (specifically related to back-end functionalities and algorithmic components), some do not address the evaluation of visual effects or UI concerns. This omission is attributable to the historical context in which these studies were conducted, since the prevalence of UI issues and visual effects in software design only gained prominence during the 1990s and 2000s, concomitant with the expansion in scale and complexity of the World Wide Web and operating systems [24].

2.3 Classification Models

This sub-chapter presents multiple methodologies implemented by previous researchers as a means to better understand the solutions and types of algorithms available to solve the problem at hand. While not all these methodologies share identical objectives with our study, there are commonalities within them and each implementation yields diverse results due to nuanced variations in algorithmic steps.

Younus Javed et al. [25] proposed a system in which Eclipse and Mozilla Firefox data is obtained, ending in a dataset of 29,000 records. They used a Bag of Words (BOW) approach to the data because it has very large dimensionality in its raw form. This method combined with feature selection techniques such as Chi-Square Testing and Term Frequency - Inverse Document Frequency (TF-IDF) algorithms result in taking the best K terms from the entire vocabulary. This results in more accuracy of the model and also in better time efficiency. Given the subset of features obtained, the researchers trained a Multinomial Naïve Bayes (NB) supervised learning model based on the assumption that any feature of a class is not related to the presence (or absence) of any other feature. The researchers found that the use of Chi-square instead of TF-IDF greatly impacted the accuracy in a positive way and that the accuracy reached its best result with a training / testing ratio of 1:11. The maximum accuracy obtained was 86%, which is a noticeable result.

Zhang et al. [18] proposed a data mining approach to classify bug types while using the Chi-square algorithm for feature extraction. On a more detailed note, the researchers mined 6,700 reports from the Bugzilla public BTS, of which they used each BR's summary, comments, and their category. These strings were then preprocessed to filter out punctuation and manually defined stop words. These reports were then manually labeled with previously defined types, so they could be separated into training and test data (to evaluate the accuracy of the models) and to prepare the training dataset for classification. The defined and used types were mentioned in 2.2. For feature extraction, the researchers mention the use of Chi-Square instead of the commonly used TF-IDF, because it measures the relationship between two variables: a feature and category. Then, supervised learning was used. The chosen model was Support Vector Machines (SVM) that achieved more than 70% prediction accuracy in most tests, with different training / test ratios and different types of model tuning. Compared to the study mentioned previously [25], Younus Javed et al. found that their own model had a much lower processing time, as the SVM (used in Zhang et al's study) [18] processing time quadratically increases as the number of documents grows.

Although not entirely within the scope of this dissertation, another study authored by Mani and Sankaran et al. [26], found that the description included in the bug report itself (which most studies in the area are based upon), involves a great amount of noisy text information, such as code snippets and stack trace details. This will inevitably affect the learning of the classifier and lead to sub-optimal outcomes. Moreover, the conventional BOW model based on this type of textual information as a means to model for feature generation fails to capture the nuanced syntactic and semantic information of the phrases and the inherent word order. To solve this, the researchers propose a bug report representation algorithm, using Attention-based Deep Bidirectional Recurrent Neural Networks (DBRNN-A). This par-

ticular model showed its capabilities to learn the context representation over large word sequences, such as those present in a bug description. Furthermore, the authors call attention to how using only the title information significantly reduced the classification performance, highlighting the importance of the BR description, at least in the context of bug triaging [26].

Tan and Liu et al. [27] studied randomly sampled bugs from Bugzilla databases in an attempt to create an automatic classifier that would analyze if a root cause of a bug is semantic, security or concurrency related. To do this, the researchers used a BOW approach in the BR summary, bug description, and discussion comments. The researchers then experimented with multiple classifiers (all of which use supervised learning on manually labeled BR), such as SVM, libSVM, Bayesian Network, and J48 decision tree. A 10-fold cross-validation was conducted on the training set (randomly sampled), to find the best parameters and classification algorithms. Through all of the layers of their implementation, the final results for the algorithm presented achieved an average F-Measure of 75.6%, a recall of 77% and a precision of 74.6%, which is noticeable. As expected, the results in these metrics vary greatly for each root cause, but it seems to be highly related to the amount of training input given to the models, e.g., the memory bugs represent 14% of the root causes in Bugzilla bugs, and it's precision was just 67%, whilst the semantic root cause represents 85.8% and their precision was of 93%.

Lo, Thung and Le [28] proposed an automated active and semi-supervised approach, based on the defect families defined by Thung in a previous study [29]. This previous study defined three defect families, which were derived from the ODC [16] scheme and included: structural, non-functional, and data flow. The researchers then used Natural Language Processing (NLP) techniques to preprocess the bug text information (title and description) in BR from JIRA public repositories. In addition to this data, since the BR analysis was *post-mortem* (after the issue was marked as solved), the changed code (bug fix) was also extracted and preprocessed using two abstract syntax trees built with a large amount of statistical data. As this type of classification uses more than two class labels, the researchers mainly used SVM Multiclass [30] in the data originating from syntax trees and textual information to train the model using previously labeled bugs. This approach got a weighted average for precision, recall, F-measure, accuracy and Area Under ROC Curve (AUC-ROC) of 0.69, 0.7, 0.692, 0.778 and 0.779, respectively. Building on their previous work, Thung et al. tried a new approach, implementing clustering (K-means), combined with active learning and semi-supervised algorithms [28]. This would result in a lesser need to create larger training datasets through manual labeling done by developers (named *Oracles*). The newer improved model used in this approach (named *LeDEx* / Learning with Diverse and Extreme Examples) achieved a precision score of 0.651, a recall of 0.669, F-measure of 0.623 and an AUC-ROC of 0.710, which outperforms by a substantial margin other state-of-the-art multiclass classification algorithms that support active learning [28]. In addition to performance improvement, to obtain these results, only 50 defects were manually labeled versus 500 from the previous study.

Huang and Persing et al. also explored the automatic generation of classifications based on ODC [16], employing a framework called AutoODC [31]. Like other researchers, this prob-

lem was framed as a supervised text classification task, but with a different approach. While typically most researchers resort to only annotating each BR with one of the predefined labels, Huang et al. not only did this annotation but also identified which words and segments within the report's summary and description contributed to the classification decision. Additionally, they employed a technique called discourse analysis to remove text segments that were considered unlikely to be relevant for classification. Although this approach might initially seem time-consuming, since the entire defect record has to be read in order to identify its category, annotation of relevant text is relatively straightforward. The resulting text was processed using the typical text tokenization, and stemming. The other unusual approach is a one-versus-other training scheme. This consists of training one classifier (SVM / NB) to determine if a report belongs to each one of the labels i.e. one classifier represents one ODC class [31]. For prediction, all models are inquired, and the one that yields the highest confidence is the one that assigns the final predicted label. The performance of this implementation was achieved via 5-fold cross-validation, showing that the overall accuracies were of 77.5% for NB and of 75.2% by SVM on a FileZilla dataset with 1 250 samples. As for the precision, recall and F-score achieved (with NB) across all classes averaged 66.73%, 78.13%, and 68,44%, respectively, although there's a very high disparity in between classes (detailed results can be seen in [31]).

With the same goal as Huang and Persing et al., Lopes and Bernardino et al. did some experiments with various types of ML models on automatic ODC categorization, including k-Nearest Neighbors, SVM, NB, Nearest Centroid, Random Forest and Recurrent Neural Networks [32]. A total of three researchers participated in manual labeling (with two of them doing an external verification) of a total of randomly selected 4 096 closed and resolved bug reports. The researchers highlighted the lack of balance between classes that would poorly bias the chosen models. In order to deal with this challenge, undersampling took place in order to decrease the number of reports in classes with an higher number of samples until balance is achieved, while the smaller classes suffered no changes. In the cases that smaller classes had too few samples for them to be relevant in the research, they were discarded. For feature extraction, several techniques were used, such as BOW, Term Frequency (TF) and TF-IDF, and for dimensionality reduction, Principal Component Analysis (PCA) was used due to its simplicity to achieve a smaller ration between number of samples and number of features (common in textual data) of two, to avoid overfitting. Although the results in performance of were not the best, specially in the broader ODC categories, such as Impact, Defect Type, and Qualifier, which averaged a accuracy, recall and precision of 35.9%, 35.9% and 37.5%, and using RNN, Linear SVM and RBF SVM, respectively. The researchers mention that this class performance was expected due to the their low quantity of data and because "as these attributes fully related to what is changed in the code, thus being the most difficult cases to classify solely based on the text of the bug reports" [32]. Even as such, the solutions presented for dealing with an unbalanced dataset in this context and the process of creation of the dataset prove to be useful for new researchers dealing with manual labeling.

Nagwani and Verma [33] have done a great amount of research in this field, but this study in particular used Bugzilla bugs combined with NLP techniques and a Suffix Tree Clus-

tering Algorithm (through an open source framework called *Carrot2*) to classify software bugs. This clustering algorithm proved to be very useful as its entropy, purity, time, and number of clusters were evaluated. Overall, the authors agree that this algorithm ensures a good way to classify bugs in small time, with good cluster purity.

Catolino et al. [7], decided to use only the bug summary / description, along with TF-IDF, Latent Dirichlet Allocation (LDA)-GA and logistic regression to achieve automatic bug classification. Given the classification scheme created (previously mentioned in 2.2), some of the authors manually analyzed, classified, and validated each assigned label in various public bug samples. In total, more than 1,000 were manually labeled. In the last iteration, the agreement on the assigned label was 96%, which is notable because the researchers have many years of experience. The authors mention that using this enhanced version of LDA (LDA-GA) eliminates the problem of setting the parameter that defines the number of topics to extract. This algorithm was used to understand the main topics in each issue for statistical analysis. The authors justified the use of TF-IDF versus other techniques such as Word2Vec, because it improved the F-Measure by 13%. For classifiers, the grid search algorithm was used to identify the best hyper parameters for each model, and of all studied, logistic regression showed the best results with an F-Measure, AUC-ROC and Matthew's Correlation Coefficient of 64%, 74% and 72%, respectively.

Mozilla, the company responsible for Bugzilla, one of the most widely used BTS, introduced a ML tool called BugBug, designed to improve bug management efficiency [34]. Although this tool is already capable of distinguishing bugs from other requests, assigning appropriate developers, identifying spam, determining the need for Quality Assurance verification, among others [35], the purpose of the following "subtool" is to automatically assign a product and component to a new reported bug (further elaborated in 3.3), streamlining bug resolution. To accomplish this, Mozilla used the vast repository of bugs manually classified by volunteers and developers over the course of two decades on their platform, and the final dataset included 100,000 bugs. Certain bugs were excluded from consideration due to insufficient data associated with their respective components, indicating a lack of historical relevance for those components. The tool makes use of the ML model XGBoost, however the specific details regarding the bugs's distribution through the various components were not disclosed. To train the model, various features were extracted from the bug titles, initial comments, and associated keywords / flags. These were pre-processed using a BOW model with 1-grams, TF-IDF and stopword removal in efforts to reduce the training phase time. Given that BugBug is already integrated into Bugzilla's platform, its results are only applied to the bug when the confidence of the model exceeds a predefined threshold that is currently set at 60%. This threshold achieves a precision rate that exceeds 80%, which is positive. According to their GitHub repository, BugBug also incorporates an automatic bug type classifier, more related to the topic of this dissertation [35]. The downside is that it currently only comprises five labels: crash, memory, power, performance, and security; however, these are planned to be expanded due to their relatively small size.

In a similar effort to BugBug, Kallis and Panichella et al. developed a tool to assign labels to GitHub project issues called Ticket Tagger [36]. Due to its requirements for porta-

bility and non-intensive so it could be deployed on a low-end server, the model chosen was the linear classifier supervised FastText on 30,000 previously labeled samples. These labels available for classification in this context were: bug report, enhancement, question, and all reprise the same ratio in the dataset. These are the labels prevalent in GitHub issue trackers. The method employed involves concatenating the issue's title and body, which is then tokenized and converted into a BOW representation, which are used as input for the training capabilities of the model. The model achieved great results across all three labels, averaging a precision, recall, and F-score of 83.2%, 82.6%, 86.6%, respectively, using a 10-fold cross validation.

Overall, from this review of the literature, we can access that many studies in this field have common points in their implementation, especially when it comes to text pre-processing and NLP, since ML models cannot understand words the same way humans do. There is a "pattern" that can be found in text classification implementations and that can be described in the following matter [37]:

1. **Data Cleaning Methods** to remove noise, and irrelevant words that won't add great amounts of information to the document. This includes stop word and noise removal, capitalisation, tokenization, lemmanization, and stemming (turns words into their most simple morphological form);

2. **Feature Extraction methods** to use as a mathematical modeling as part of a classifier, usually by calculating the amount of times words appear in a document, relating them to how relevant they might be, or other statistical methods. These include BOW, TF-IDF, Word2Vec, FastText, and GloVe;

3. **Dimensionality reduction methods** to retain only the most important information in a document, and to help reduce the size and computational cost of training the models with big feature spaces in the datasets. These methods include PCA, Linear Discriminant Analysis (LiDA), non-negative matrix factorization, Autoencoder, random projection and others;

4. **Classification algorithms** that usually use supervised learning, such as NB, K-Nearest Neighbor, SVM, decision trees, random forests, or deep learning classifiers as deep neural networks, recurrent neural networks, and others;

As for semi-supervised or unsupervised approaches, not very common in the field of bug classification, when used in research of other problems involving text classification usually include clustering techniques to exploit the existence of small portions of labeled versus unlabeled data by clustering similar points together. Then it is assumed that all points in a cluster share the same label if there are some labeled data present within it [37]. Most researchers tend to employ supervised learning approaches, even in cases where they encounter limited quantities of labeled data or where manual labeling is necessary.

2.3.1 Emerging Methods with LLMs

With the very recent rise of Large Language Models (LLM) (such as OpenAI's Generative Pre-training Transformer (GPT) [38] and Google's BERT [5]), and their accessibility, some of the mentioned text classification and NLP techniques could have become obsolete, as these models can achieve better performance than most statistical models, due to the large amount of text data on which they have been trained [39]. These tools specialize in generating coherent and contextually relevant text, and their capabilities have been proven in fields such as text analysis (especially sentimental) [39].

In 2022, Mohammed Siddiq and Joanna Santos studied the possibility of using a BERT-based classification technique to automatically classify real-world GitHub issues as bugs, questions, or enhancements [40]. They extracted more than 700 000 labeled issue reports and used 80 518 in their test set to evaluate the performance of the proposed solution, which ended up achieving a remarkable 0.8586 F1 score, with an average of 0.8571. To obtain these results, the researchers performed data preprocessing, in which text cleaning and feature extraction are included due to the high number of code segments in the issue descriptions, as well as white space and line breaks. Then, as BERT is a pre-trained model, it requires data in a specific format, so that transformation was done (bert-base-uncased). This pre-trained model was chosen because it only takes in lowercase characters. For training, the AdamW stochastic method was used and its results were very positive.

In another type of implementation in the search to improve bug-related software quality, a study was carried out in which a method was created to predict the severity of the problem using source code metrics and LLM [41]. In practice, this was done through analysis of more than 3 000 buggy methods from open source projects, and public code metrics (such as lines of code, cyclomatic complexity, nested block depth, and others). These were used to train various classic machine learning models, such as SVM, Naive Bayes, Decision Trees, and others. The researchers also used the pre-trained LLM called CodeBERT which was trained on programming language and natural language inputs, and as such fit the needs for the algorithm implementation, although no bug descriptions were mined in the dataset, as commonly used in other studies in this field. As the Random Forest model showed the best results, even though no hyperparameter tuning was performed, the model was chosen and used to compare performance versus the CodeBERT model, and the LLM outperforms it by 29%, 33% and 140% for F1-weighted, AUC-ROC and Matthews Correlation Coefficient, respectively. These numbers show just how much improvement LLM can provide to ML studies.

In an implementation to detect buggy code, based on a code pair classification task, Kamel Alrashed tested various models, with differences in how they were fine-tuned [42]. In this in-context approach, the LLMs (GPT-3.5 and CodeLlama) are prompted with a buggy version of code and its fixed version, instructing the model to select the buggy one. The difference in model performance with the code-pair classification was very impressive compared to a common binary classification approach, as the accuracy increased from 54.15% to 72.93% for GPT-3.5, and from 50% to 69.87% for CodeLlaMa. The F1 scores were 84.34% and 82.26%, respectively, which is notable. Compared to the supervised learning approach used with CodeBERT and CodeT5, directly fine-tuned with different batch sizes, learning rates,

and number of epochs, the results are even more noticeable, as these two models had very low performance, as they both guessed correctly 50% of the time (closer to random guessing), which the researcher attributes to the small dataset through which they were tuned.

2.4 Conclusion

This chapter provided a review of the state of the art in bug classification machine learning models and bug taxonomy classification schemes. Each study discussed provided insight into the development of prevalent methodologies, highlighting potential avenues for future research and implementation. The review offers a balanced assessment of the strengths and limitations of the existing classification frameworks, highlighting the practice of reusing datasets with highly generalized labels, which provide little information about the issue itself. The creation of new classification datasets to be used with other models frequently replicates this issue, resulting in tools that fail to provide meaningful insights to developers about the bugs reported. There is a clear need for a tool capable of categorizing issues previously identified as bugs into more informative subcategories, enhancing the context and information passed on to developers. Furthermore, this area of study has garnered significant academic interest due to its importance for software quality, even more so with the recent rapid rise of LLM to facilitate NLP tasks.

The results of some of the classification ML implementations mentioned can be seen in Table 2.1, to compress and simplify the performance analysis [1].

Researchers	Information used	Pre-processing	Learning Type	ML Model	# of Bugs	# of Labels	Accuracy	Precision	Recall	AUC-ROC	F1 Score
Santos et al. [40]	Issue Title + Body	Text cleaning	Supervised	BERT (bert-based-uncased)	700,000	3	-	0.8586 (avg)	0.8586 (avg)	-	0.8586 (avg)
Thung et al. [29]	Issue Title + Description + Bug Fix + Code Metrics	Tokenization + SW Removal + Stemming	Supervised	SVMMulticlass	500	3	0.778	0.69	0.7	0.779	0.692
Lo et al. [28]	Issue Title + Description + Bug Fixing Code + Code Metrics	Tokenization + SW Removal + Stemming	Active and Semi-supervised (Self-training)	K-means + SVMMulti-class	50	3	-	0.651	0.669	0.71	0.623
Catolino et al. [7]	Issue Summary + Description	TF-IDF + LDA-GA	Supervised	Logistic Regression	1,000	8	-	-	-	0.74	0.64
Hemmati et al. [41]	Code metrics	-	Supervised	CodeBERT	3,000	4	-	0.71	0.72	0.87	0.70
Tan et al. [27]	Issue Summary + Description + Comments	BOW	Supervised	SVM + Bayesian Network + J48 Decision Tree	2,060	3	-	0.74 (avg)	0.77 (avg)	-	0.75 (avg)
Zhang et al. [18]	Issue Summary + Comments + Category	Chi-Square	Supervised	SVM	6,700	7	0.7	-	-	-	-
Kallis et al. [36]	Issue Title + Description	Tokenization + BOW	Supervised	fastText	30,000	3	-	0.832	0.826	-	0.8663
BugBug [34]	Issue Title + First Comment + Keywords / Flags	BOW + TF-IDF + SW Removal	Supervised	XGBoost	100,000	NA	-	>0.8	-	-	-
Persing et al.[31]	Issue Summary + Description	Discourse analysis + Stemming + Tokenization	Supervised (One-vs-Other)	Naive Bayes	1,250	14	0.775	0.667	0.781	-	0.6844

Table 2.1: Performance summary of some of the bug classification approaches across multiple contexts using machine learning, as described in literature review, accompanied by their respective performance metrics and types of algorithms.

Chapter 3

Methodology

3.1 Introduction

Most studies on bug classification and prediction (even if not entirely aligned with the present research) rely on prelabeled datasets. However, it is crucial to recognize that categorizations can vary greatly depending on the observer's perspective (further discussed in 3.5). As a result, the model's perspective, acquired through training, can differ from the original intent and context of the bug reporter, particularly in unseen data. This discrepancy can impact the relevance of the model's results [13].

To address the high complexity of textual data, this study employs supervised learning techniques, which have proven effectiveness in similar datasets and intents, as described in Section 2.3. This methodology, achieved with the technologies presented in Section 3.2, is presented in multiple stages, which include: an overview of data pre-processing and extraction (Section 3.3); a description of the types of category that make up the labeling system (Section 3.4); and the dataset creation process (Section 3.5); and a detailed explanation of the machine learning process and the evaluation techniques used (Sections 3.6 and 3.7, respectively).

3.2 Technologies Used

Although the list of technologies used throughout this implementation is relatively concise, this subsection provides a detailed and comprehensive examination of the most relevant ones.

3.2.1 Python

Python is an open-source high-level interpreted language, and is considered general-purposed even though its use is most popular in ML and Data Science implementations, due to the rich ecosystem of free libraries and frameworks that accelerate the development [43] [44]. It was created by Guido van Rossum in 1991 and is characterized by the peculiar way code is written. The logic blocks are written and defined through white space indentation, resulting in very clean code. Its versatility with multiple different technologies and ease of learning and use keep it as a top option for programmers despite its drawbacks, like slowness and large memory consumption related to its dynamic typing nature. The full code of this methodology was developed with Python version 3.10.2, using Visual Studio Code, a lightweight Integrated Development Environment created by Microsoft. This tool enhances productivity through its combination of features, including support for extensions such as

Git for version control and IntelliSense for intelligent code completion. In addition, its integrated debugging UI and terminals greatly contributed to the development process.

3.2.2 Sklearn

Scikit-learn is a widely used Python open-source library for ML that provides simple and efficient tools for data analysis and modeling. Built on top of NumPy, SciPy, and Matplotlib, this library contains a very extensive and well documented list of different algorithms for classification, regression, and others, which functions as a very good foundation for very fast implementations and evaluations in those fields [45]. In this implementation, this tool was used mainly for evaluations, described more extensively in Section 3.7.

3.2.3 PyTorch

PyTorch is one more open source deep learning framework developed by Facebook, known for its flexibility in building deep learning models, such as those used in language processing. This framework facilitated efficient utilization of the full computational power of available Nvidia GPUs through CUDA support. Using both the framework and CUDA, developers can run implementations across multiple GPUs in parallel, significantly accelerating the computation process associated with model training.

3.2.4 Transformers

The Transformers library, developed by Hugging Face, is a robust and comprehensive toolkit specifically designed for engaging with state-of-the-art NLP models. This open-source platform gained popularity and recognition for democratizing access to advanced NLP technologies and resources. It is particularly used for its extensive collection of pre-trained transformer models, including widely utilized architectures such as BERT and Meta's LLaMA. Its usaeg is explained further in Section 3.6.2.

3.3 Data Extraction

In the field of data mining, it is important to keep in mind the quality of the data under study due to its consequence in determining the efficiency and reliability of the prediction model. Poor data quality leads to unexpected negative details of predictive results. To ensure the credibility of the model, it is also important to use real-world data, which can be extracted using the most widely used BTS, such as JIRA and Bugzilla. These platforms have hundreds of thousands of BRs, and some companies (such as Eclipse, Apache Foundation, Mozilla and JIRA itself), have their bug reports open to the public, as some of their software is open-source, and to allow clients to report their biggest problems e.g: Mozilla Bugzilla's Website 3.2.

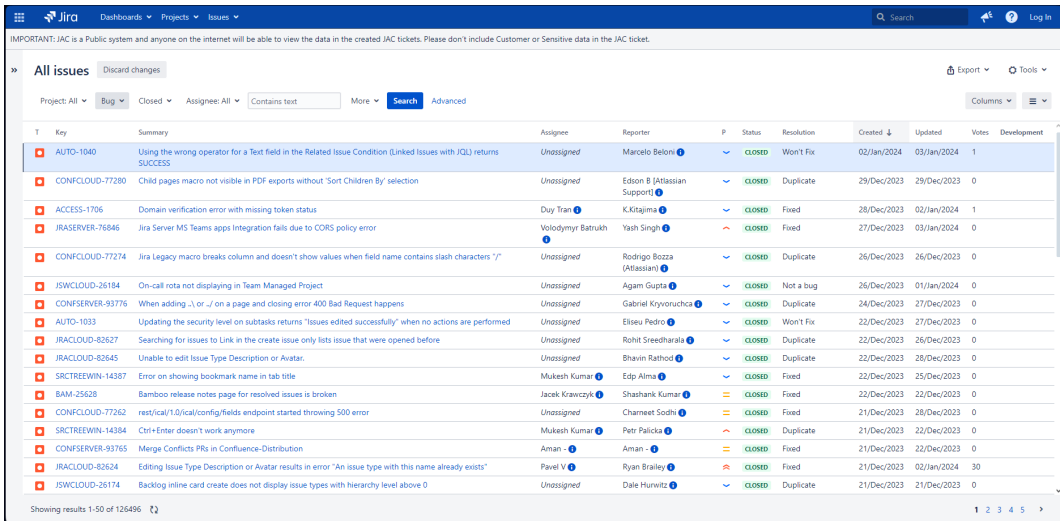


Figure 3.1: Atlassian's JIRA public dashboard. Available at: `https://jira.atlassian.com/issues/`

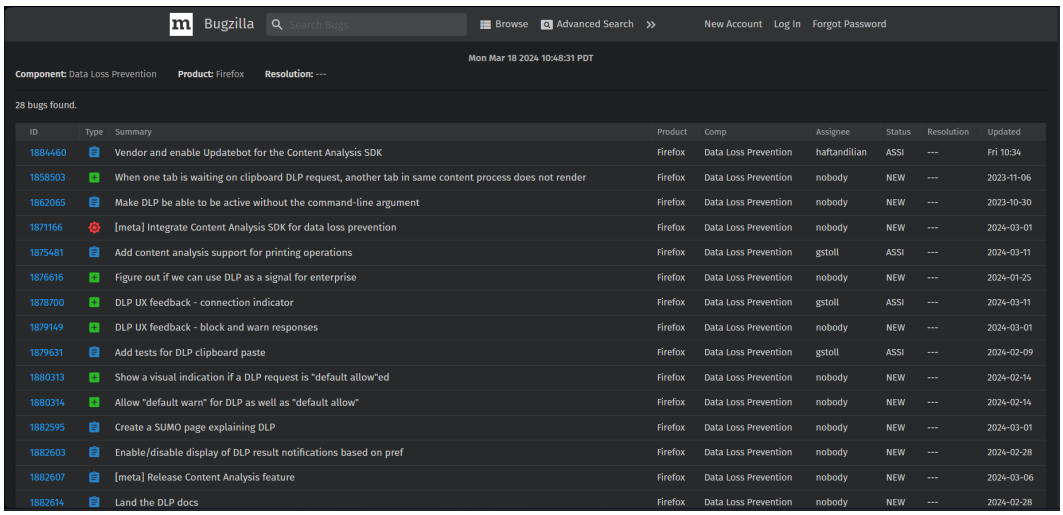


Figure 3.2: Mozilla's Bugzilla public dashboard. Available at: `https://bugzilla.mozilla.org/`

These tools also provide well-documented public APIs that allow queries of dozens of different projects and types of bug reports that we want to extract. As observable in figures 3.1 and 3.2, every issue reported to these BTS has a different set of categorical and non-categorical (free text) attributes that characterize them and can be filtered until the appropriate results are returned. These attributes are very helpful and include the following: *id*, *assigned_to*, *component* (to which the issue belongs), *status*, *severity*, *url*, *summary*, *keywords*, *dupe_of* (if it is duplicate), *estimated_time* (devoted to fixing the bug), and many others [46]. In these, an attribute called *classification* is also present, and according to the API documentation it "tends to be used in order to group several related products into one distinct entity" [47], however, in real-world scenarios, this classification seems to fall short and lack the amount of information that could be given to bug-fixing developers, i.e., in Mozilla's case in which the Figure 3.3 shows the only classifications available.

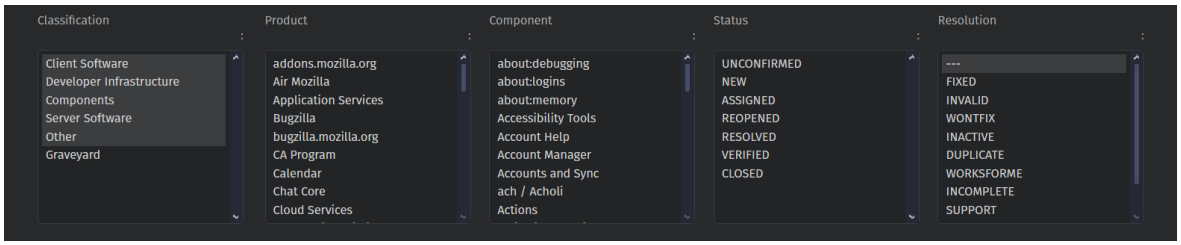


Figure 3.3: Mozilla's Bugzilla potential categorical attribute values. Available at: `https://bugzilla.mozilla.org/query.cgi?format=advanced`

Since this research is based on textual descriptions made by the reporter on the issue, the relevant (and used) attributes are id, summary, and status. Of all the statuses available as options to filter out issues (which can be observed in 3.3, the most relevant one for the research was "CLOSED" (although not restrictive), as it defined the end of the bug lifecycle. Regarding the type of issue, it was defined in the API request as a "defect", since improvements and other types of issue do not have any relevance to the algorithm.

As for Atlassian's JIRA, a Python library was used, which makes the interaction with the API a lot easier. Through it, issues were filtered by type "Bug" and all projects available in the organization's JIRA were selected, hence the empty string in Listing 3.1, to ensure the most variety possible in the types of bugs that could be found.

Listing 3.1: JIRA library usage to filter issues.

```
from jira import JIRA
jiraConnection = JIRA('https://jira.atlassian.com')
issues = jiraConnection.search_issues(f'project="" AND type=Bug',
    startAt=start_at, maxResults={max_results})
```

3.4 Classification Scheme Creation

Of the studies reviewed, most classification schemes sound incomplete, either due to the historical context in which they were created or simplified for research purposes, and therefore are not entirely accurate for real-world problems, as they seem to lack useful and detailed information. The classes are too vague and include a great number of similar (although different) types of bugs. It can be hard to assign only one category to each bug as all software is dependent on itself, but the categories defined in these studies seem to have little relevance in a real project in which the bug triaging / analysis process would not even need human interaction. For example, it would be very helpful for quality assurance teams if a reported bug could be correctly detected as a memory allocation problem and then automatically assigned to the debugger with the most experience in the field or to the person responsible for those types of tasks in the software project.

Keeping the previously mentioned classification scheme by Freimund et al. in mind [15], it is possible to create a new scheme that is generic and extensive enough to be used by most software developers, regardless of the architecture of the software that has been built. Ideally, this classification scheme should be technical enough to be objective for developers and

provide a great deal of examples due to the difficulty in one-class classification mentioned in previous sections. Taking into account the strengths of the research and the schemes mentioned in Table 2.2, and keeping their weaknesses in mind, the result of the proposed classification scheme as shown in 3.1. With the scheme, some examples are given to each subtype as way to clarify the concepts.
This scheme allows for a more complex and precise description of each bug, giving developers more information, and its use can be more versatile in future studies of automatic bug triage. Given the involvement of multiple classes, it was expected that the efficiency and performance of the model would suffer negatively. To preempt this, a hierarchical classification approach was thought out, fragmenting a complete multi-class problem into a set of smaller classification problems. This strategy ensures that even if the performance / confidence of the model in each sublabel is low, it would still provide developers some relevant and accurate information confidently, although more generalized. For example, while the model might correctly identify a bug related to GUI as interface-related, pinpointing its exact subtype, i.e. a navigation-related issue, might be trickier due to the higher number of classes dedicated to the GUI, and possible similarities in the text within the subtypes. Still, knowing that it is a GUI-related bug is valuable information for developers, even if the finer details are not clear. It is preferable to have accurate information for comprehensive bug analysis rather than a plethora of uncertain and not-confident / reliable details. In addition, given the subjective nature of software performance, where resource consumption can be justified by program requirements, and user perceptions of acceptable wait times may be skewed, adding more granular subtypes to this main category proved to be counterproductive. Furthermore, many performance-related issues reported are often symptomatic of other underlying problems, making finer categorization less effective in addressing the root causes.

3.5 Data labeling

The development of a new classification scheme implies that there must be a meticulous data labeling process, aligned with the classes within the scheme. Manual data labeling requires a substantial time investment and is very repetitive and prone to errors, especially since it demands great attention to detail. Many researchers mentioned the benefit of collaborative efforts within research teams, which greatly alleviated time constraints, but also mitigated the likelihood of biased classifications, as there was a comparative analysis between all members. These analysis created discussions until a consensus (or majority approved) label was reached [29] [13] [7]. This collaborative nature of the manual labeling process also has advantages from the perspective of diversity of expertise. Each researcher who participates in the discussion typically has unique experience with various technologies for different amounts of time, thus enriching the discourse with diverse perspectives. This diversity enables the exploration of multifaceted viewpoints, potentially shedding light on nuanced aspects of bug classification that might otherwise have been overlooked. Given the lack of human resources for the author to engage in collaborative discussions, this limitation was addressed by leveraging the capabilities of the LLM ChatGPT 4 and 3.5, in an attempt to

Main Type	Subtype	Definitions	Example
Security	Authorization	Wrong user access authorizations and privileges, and other security credential issues.	Users can access my project details through the API even when settings say it's private
	System Misconfigurations	Errors in tuning of compatibility that prevent different softwares of working together.	The installation package doesn't ask the OS for the required permissions
	Vulnerabilities	Errors detailing security breaches that can be exploit to detriment of the software and its data.	Chart component allows XSS through project name filter
Code Logic	Calculation	Wrong value calculations due to wrong formulas and other counting issues.	Time left in component showing 2 more hours due to wrong timezone
	Functional	Wrong iterative and conditional blocks of code, return values, unexpected crashes and other functional errors.	The program deleted a non-selected project
	Internal Search Engine	Issues with misleading and wrong search results, queries and suggestions.	Search bar returns 0 results when index 41600 is reached
	Libraries	Issues related to the sub-optimal utilization of external libraries, whether due to programming or the usage of outdated versions.	Third party form submitting package is incompatible with version 4.12
	API Integration	Errors that include misleading API endpoints or response codes, as well as unexpected responses or required parameters.	/projects GET endpoint returns wrong HTTP code but returns correct data
Data	Incorrect Data Structure	Issues with data types defined as expected versus found.	Database constraints are not well defined
	Memory Faults	Errors related to poor memory management and allocation.	Out of memory exception when uploading file under size requirements
	Pointer Faults	Exceptions caused by wrong pointer usage, including their incorrect nullity and inappropriate manipulation.	NPE when creating a new context field on issue navigator
	Data Transmission	Errors related to unrelated, corrupted or unusable received / sent data.	The input on my forms does not correlate to what the software uses
	Encoding	Issues with text encoding / serialization schemes and other special characters.	Can't create username if input has special characters
GUI	Navigation	Errors in changing views, broken links and pagination issues.	When I click on my project, the settings button links to a 404 page.
	Style	Errors in styling, responsiveness of the graphics, resizing, and device compatibility.	If I have too many projects in my profile list, it's not possible to click on the delete button
	Validations	Errors in the validation of incorrect input data.	Admin form submit button is greyed out and doesn't respond even though everythings correctly filled
	Content	Errors in the display of information, incorrect / misleading indications in inputs / buttons and other HTML elements and their functionality.	Delete icon is the same as the submit icon, leading to errors
	Accessibility	Errors related to language and translation options, missing alt text, and other missing Web Content Accessibility Guidelines.	Spanish month translations on the date picker is wrong
Performance	NA	Errors related to code optimization, resource utilization and timing issues.	Rebasing a project takes 15 minutes of load time and the process is at almost 100% memory usage

Table 3.1: Proposed Bug Classification Scheme, with the definitions and examples for each type/subtype.

lower classification bias.

3.5.1 ChatGPT for labeling assistance

ChatGPT is a state-of-the-art LLM tool based on the GPT (transformer) architecture, developed by OpenAI. This model can generate human-like coherent text (and images) based on input prompts given in text format. This is done through a deep learning approach to understand (and replicate) natural language, which revolutionized the field. This model uses unidirectional self-attention mechanisms that weigh the importance of different words in a sentence through their internal relationships, from multiple perspectives, and was pre-trained in various internet text data using unsupervised learning. It was then fine-tuned with supervised learning techniques to improve its performance in replicating complex text. Both ChatGPT 3 and 4 (the ones used in this research) have been trained on 175 billion and 100 trillion parameters, respectively [48].

ChatGPT has proven a wide range of capabilities, mainly due to the large amount of text and image data used in its training, including content generation (i.e. creative writing, image and report generation), use in conversational agents, as an educational tool of support, and others. Although it ”understands” text, it does not possess genuine understanding, so its responses are based on patterns in the training data, which means the answers can be biased, incorrect, or nonsensical. This was one of the challenges when trying to use this tool to support the data labeling process.

For prompting, the initial classes were introduced to the model either by image representation for ChatGPT-4 or textual input for ChatGPT-3. Subsequently, the description and title were provided as input, and the model was asked to predict the type and subtype of that sample. An example of a prompt follows: ”Given the bug types for classification, how would you classify this bug?” followed by the textual data of the BR. When the result was accurate (given the context of the study), it revealed itself to be important because it added another layer of valuable knowledge and perspective, enhancing the classification process beyond the label assigned by a human with few years of experience in software development. This additional layer of input was particularly useful for resolving ambiguities in bug reports, where the model’s predicted labels and justification for those choices helped clarify the intent of the bug and addressed conflicts arising from (what initially seemed) multiple possible types.

However, several limitations were encountered during the use of this tool. Financial constraints and prompt quantity limitations within ChatGPT-4 led to delays and a slow labeling process. Moreover, commonly it was clear that the model did not grasp the objective of the task at hand, especially in the aspect of struggling to assign multiple labels / hierarchical classification consistently. In such cases, an example of the intention as given in hopes to try to clarify the goal in a way that resembles one-shot classification, as this model is usually successful in rapid adaptation in one-shot and few-shot tasks [49]. However, this aggravated the problem of prompt quantity, as additional prompts were needed to maintain the classification accuracy and the model within the limits of the task. This was also exacerbated by the limitation on the models’ tendency to misinterpret the context of the bug reports. For example, bug reports related to libraries were frequently misclassified as Code Logic and

of the Library subtype, overlooking the possibility (which was common) that the project in which the bug report emerged could be the library itself, and not its usage through other developers. Taking this into account, the labels given by ChatGPT (with multiple prompts to stay within the context) differed from those humanly assigned by 37%, which is a significant percentage even with the amount of ”help” provided to the model. Despite these challenges, cross-checking the results generated by ChatGPT with human labels proved beneficial, although it slowed the labeling process due to quality control evaluation, to try to achieve a high quality dataset.

3.5.2 Class distributions

As witnessed in previous investigations (referenced in 2), a notable but predictable challenge arises from the considerable class imbalance, attributable to the inherent discrepancies among various types of bugs. Some types of issue are simply more likely to be reported. It is also important to acknowledge the potential bias introduced by limitations on the expertise of the individual author in software development. During the initial phases of the manual labeling process, the selection of issues for analysis was performed by random sampling. Each selected BR underwent inspection criteria, including a complete understanding of the author and the presence of relevant information within the textual descriptions provided by the author of the BR, excluding comments from developers. Although some of the developer’s comments have key information about how the issue was solved, their removal was a deliberate decision made after the analysis of a great number of bugs. Text data is highly complex and hard to filter, and a great number of comments included developers discussing categories, tagging each other for notification / assignment purposes, and reporters thanking them / having other informal / non-relevant interactions. This approach aimed to uphold the integrity of the data set by ensuring the inclusion of only pertinent information and information that was very clear on the reason behind the bug that would otherwise severely affect the quality of the learning algorithm if not filtered out. This method also excluded issues not related to bugs (such as documentation and new functionality-related issues), as well as those that were commonly misclassified, not in English, were clearly issues created only to test the BTS, or relied solely on screenshots or different logs/crash reports. Subsequently, the issues that met these criteria were labeled with the assistance of the insights given by ChatGPT as mentioned in 3.5.1. Bugs with associated duplicates (often referenced through links or comments) were significant, as they described the same problem from various textual perspectives and descriptions. This diversity facilitated easier labeling by providing multiple points of view on the issue. The class distribution on this random sampling of 522 issues (non-bugs included) is shown in Tables 3.2 and 3.3, and Figure 3.4.

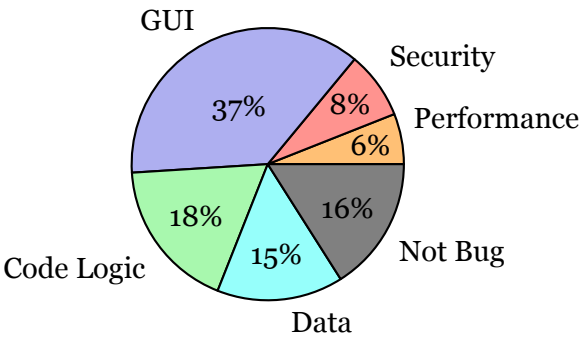


Figure 3.4: Issue distribution through main types achieved through random sampling.

Main types	Amount of issues
Graphical User Interface	195
Code Logic	93
Data	76
Security	42
Performance	30

Table 3.2: Issue distribution through main types achieved through random sampling (Non-bugs not included).

Main Type	Subtype	Amount of Bugs
Security	Authorization	18
	System Misconfigurations	12
	Vulnerabilities	12
Code Logic	Calculation	9
	Functional	40
	Internal Search Engine	8
	Libraries	18
	API Integration	18
Data	Incorrect Data Structure	14
	Memory Faults	10
	Pointer Faults	10
	Data Transmission	31
	Encoding	11
GUI	Navigation	47
	Style	54
	Validations	16
	Content	66
	Accessibility	12
Performance	NA	NA

Table 3.3: Bug Distribution per subtypes.

Through the examination of the class distribution observed so far, it had become apparent that an imbalance in classes was inevitable. This phenomenon arose from several factors, including the likelihood that certain types of bug are reported more frequently than others, as well as the inherent bias introduced by the requirement for the author to possess a comprehensive understanding of the bug and its context for it to be selected for labeling. For instance, the author found GUI bugs more accessible to comprehend due to their typically straightforward descriptions, compared to Code Logic bugs, which often need a significant amount of contextual understanding of the software to which they belong. Another observation to note is the percentage of non-bug issues that were identified (~16%). These include

”bug” reports asking for enhancements, new features, asking for different user privileges or accesses, asking for system deletions, compilations of issues, BTS tests, and others. Although this percentage may not be as substantial as reported by other researchers (mentioned in Section 2.3), it still exerts a significant negative impact on the dataset.

Later, when the class imbalance started to have a severe impact on research, because some labels had so few examples that the algorithm would not even consider them, some issues were searched by hand based on keywords that were likely to be found in the description of bugs of that category. Using the research conducted by Catolino et al., which delineated various topics within each bug category, as demonstrated in Table 3.4 [7], a generalized understanding of the approach was facilitated. Despite the differences in the classification scheme proposed in this dissertation, their insights proved invaluable in shaping this aspect of the methodology.

Categories	Topics
Configuration issue	link, file, build, plugin, jdk
Network issue	server, connection, slow, exchange
Database-related issue	database, sql, connection
GUI-related issue	page, render, select, view, font
Performance issue	thread, infinite, loop, memory
Permission / deprecation issue	deprecated, plugin, goal
Security issue	security, xml, packageaccess, vulnerable
Program anomaly issue	error, file, crash, exception
Test code-related issue	fail, test, retry

Table 3.4: Relevant topics by bug type [7].

To combat the low sample sizes in these classes, oversampling was used, as the websites of the BTS were browsed based on keywords that could be related to that class, based on their examples. Such keywords are divided by subtype in Table 3.5, and even though sometimes these queries returned issues as ”bugs”, those that were not actually bug related were ignored. To ensure consistency and ease of reading, the color coding used in Table 3.5 is identical to that in Table 3.3. In the context of this study, the ”Performance” type does not encompass any subcategories, meaning that its associated keywords represent the class in its entirety. These keywords were essential to address class imbalances within the dataset that was previously randomly sampled. Although some keywords are shared across various categories, highlighting the inherent intertwining of software and occasional ambiguity in bug classification, analyzing the full context and intent of each bug ensured that the use of these keywords greatly facilitated the bug search process. As such, when queries on the respective website returned relevant bugs given some of these keywords, they were analyzed and added to the dataset with the most relevant type and subtype label.

Subtype	Keywords
Authorizations / Permissions	password, permission, role, access, restricted
System Misconfigurations	configuration, mismatch, compatability, setup, default, settings
Vulnerabilities	attack, attacker, leak, vulnerability, backdoor, brute force, malware, exploit, plain-text
Calculation	calculation, wrong, value, results, bad, number, time, count
Functional	NA
Internal search engine	query, results, inconsistency, SQL, search, wrong
Libraries	library, deprecated, outdated, upgrade, incompatible
API Integration	http, REST, API, error code, status code, endpoint, request, response, payload, JSON, header, timeout
Incorrect Data Structure	wrong type, response, expected, typeerror, incompatible
Memory faults	buffer, overflow, memory, leak, exception, RAM, thread
Pointer faults	pointer, null, NPE, exception
Data Transmission	transmission, storage, format, export, corruption, missing
Encoding	base64, ASCII, characters, encoding, transmission, UTF-8
Navigation	hyperlink, click, nav, navigation, key, press
Style	wrong, style, color, shows, overflow, fit
Validations	check, number, string, format, submit, form, input, indication
Content	content, wrong, results, shows, label, displayed
Accessibility	language, options, keybind, shortcut, translation, input
Performance	infinite, complexity, 100%, o(n), o(n^2), slow, fast, optimization, metrics, benchmark

Table 3.5: Keywords per subtype used in manual sampling in order to balance dataset class distribution.

The subtype ”Functional” represents the most extensive category among the identified error subtypes. Consequently, no targeted searching was conducted for this sublabel, as the process was straightforward, which means there was no necessity to utilize specific keywords.

At the end of the labeling and oversampling process, the dataset comprised of a total of 1047 manually labeled bug reports. This dataset exhibited a more balanced distribution across bug categories, as illustrated in Figure 3.5, and Tables 3.6 and 3.7. The class ”Performance” has the lowest number of bugs labeled mainly because no subtypes were defined, which decreased the need for bugs, since this category was only used in one of the models, which will be explained in detail in subchapter 3.6.2.

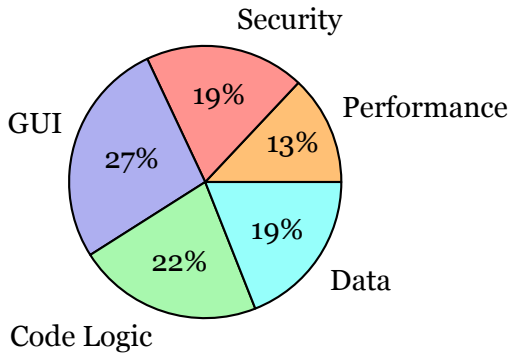


Figure 3.5: Bug distribution through main types achieved through random and manual sampling.

Main types	Amount of issues
Graphical User Interface	280
Code Logic	232
Security	203
Data	200
Performance	132

Table 3.6: Bug distribution through main types achieved through random and manual sampling.

Main Type	Subtype	Amount of Bugs
Security	Authorization	76
	System Misconfigurations	42
	Vulnerabilities	85
Code Logic	Calculation	47
	Functional	51
	Internal Search Engine	50
	Libraries	39
	API Integration	45
Data	Incorrect Data Structure	27
	Memory Faults	36
	Pointer Faults	39
	Data Transmission	50
	Encoding	48
GUI	Navigation	56
	Style	56
	Validations	52
	Content	72
	Accessibility	44
Performance	NA	NA

Table 3.7: Bug Distribution per subtype after random and manual sampling.

To ensure bug report variety through usage of real world data, all projects provided from the BTS were considered for labeling, reprising of more than 150 different software projects used by thousands, if not millions of clients on a daily basis, such as Atlassian's Fisheye, Trello and Confluence, Mozilla's Firefox and Thunderbird, and Eclipse Foundation's Jakarta and Eclipse IDE. Even though no projects were filtered out, the dataset includes samples of 89 different projects. The five projects with most labeled bugs in the dataset are presented in Table 3.8.

Project Key	BTS
JRASERVER	JIRA
CONFSERVER	JIRA
JRACLOUD	JIRA
SRCTREEWIN	JIRA
CONFCLOUD	JIRA

Table 3.8: Top 5 Projects with most labeled bugs in the dataset.

3.6 Algorithm

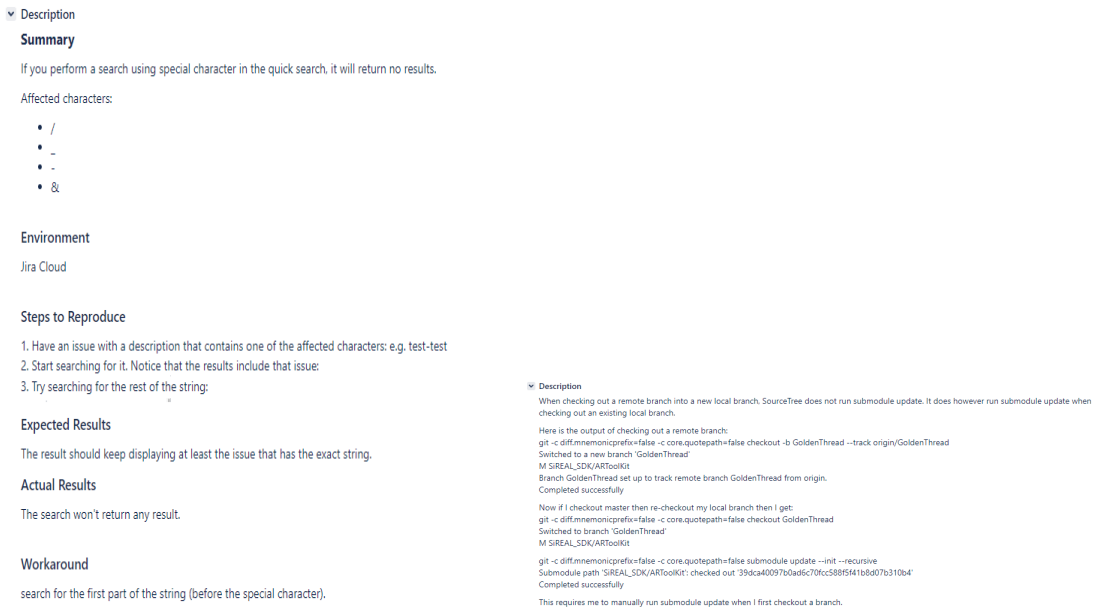
After analyzing the literature on bug classification, the presented algorithm shows a new approach in the field and uses some new techniques used in other fields of text classification,

but never applied in combination to these software bug reports. It is important to separate the different processes into various subsections in order to discuss their design, implementation, and rationale behind the chosen approach as a whole. Since the main topic of the problem to be solved is how to input text into a ML algorithm correctly in order to achieve the best results possible, we start by explaining the text pre-processing.

3.6.1 Text Pre-Processing

When dealing with a large amount of text that will be used in ML, there has to exist some form of document pre-processing, as models cannot understand words the same way humans do. To do this, we have to use NLP technologies on the previously extracted text. As mentioned in the previous chapter, most researchers do data cleaning (which involves stopword removal, tokenization, lemmanization and stemming), and then TF-IDF to analyze the significance of a string in a document. This turns a normal human-written text document into a text with reduced complexity, since all words are lowercase, punctuation is removed, and words are turned into their most simple morphological form. To enhance comprehension, an example string before NLP: ”Whenever I click the ’Submit’ button on the checkout page, the application crashes and shows an error. I encountered this problem while using the app on my Windows PC”. After NLP techniques, that same string would now be ”click submit button checkout page application crash show error encounter problem use app windows pc”. Then, TF-IDF vectorization turns the processed strings into numerical vectors that can be used in ML algorithms by showing how important a word is to a document, and result in some complexity and dimensionality reduction so the model’s training isn’t as computational expensive. However, in the present methodology, a different model was used (BERT, more information is given in Section 3.6.2), which does not require the application of some of these techniques.

Upon analyzing several bug reports, it became evident that while developers and reporters made efforts to maintain some level of structure in noncategorical bugs, there exists a considerable number of bugs with free-form text or structured in varying orders, and this applies to Bugzilla and JIRA. The downside of having free-text inputs in the description to ensure its flexibility is the lack of consistency of the form of bug reports, which could greatly impact studies done on these reports. In addition to the lack of structure, some reports include various logs, code snippets without the proper formatting, and *code* tags. Some examples of descriptions are given in Figure 3.6.



(a) Bug JRACLOUD-71066 - a well-structured bug report.

(b) Bug SRCTREE-1097 - a poorly structured bug report.

Figure 3.6: Comparison of different bug report structures.

As observable in 3.6, although all of JIRA's descriptions belong to a free-text attribute called "Description", some users divide their text into sections such as "Summary", "Workaround", "Environment", "Steps to reproduce", "Steps to replicate", "Observations", "Reproducible", "Is reproducible?", "Expected Results", "Actual Results", and others, while some users choose not to structure their text at all. These headers exhibit variations in their formatting, originating from definitions written manually by users. For example, headers may differ in their use of bold formatting, some of which are bolded, while others are not. Additionally, some headers may be designated with the `h3` tag, while others utilize the `h2` tag, and so on. Furthermore, variations in capitalization are evident, with some headers beginning with capital letters and others do not. This is explained with the possibility of allowing users to format their reports however they feel most comfortable, whether it be HTML tags or Markdown language. In efforts to clean up the text of each description to obtain only the relevant textual descriptions that could provide important information to the model, the following actions were taken:

1. The `.xlsx` labeled dataset file was loaded into Python with the help of the `pandas` library. Each row in the dataset contains the bug identification key (column `bugId`), the type and subtype given by ChatGPT (`Type LLM` and `subtype LLM`), and the type and subtype given by the researcher (`Type Res` and `subtype Res`) , and the BTS where the bug was found. As such, each BugId and BTS value was used to search for the bug sections in files, divided by project key that were extracted automatically throughout the project, to be able to return their descriptions. When the bugs could not be found in these files, they were searched by their key using their relevant API. Using files with a big number of pre-fetched bugs helped reduce computing and waiting times between iterations, dodging repeated API calls, especially in the earlier stages of the research im-

plementation, where multiple experiments were made in small amounts of time. The textual, key and project data was joined in lists of dictionaries for ease of use.

2. **For JIRA issues:** To go over the hurdle of the different tags in the textual data to signalize different headers, all the capital letters in the headers mentioned above were considered. Besides that, an assumption was made that if a description started with a header, then it would be likely for the description to be somewhat structured, and as such all the text in between the first two headers would be relevant. If no starting header was used, then the relevant text would start at index zero of the string, and its end would be considered be the position the next header found or to the full length of the description. There were also multiple cases where no header was used, but strings similar to ”NOTE. This suggestion is for JIRA Cloud. Using JIRA Server? See the corresponding suggestion panel” or ”NOTE. This bug report is for JIRA Server. Using JIRA Cloud? See the corresponding bug report panel”, were present to guide users that might be searching for bug reports related to the wrong product, show up at the start of the description. These were removed;

   For Bugzilla issues: A large percentage of BR descriptions start with a description of a user agent and / or Build Identifier e.g. ”User Agent: Mozilla/5.0 (Windows NT 6.1; Win64; x64; rv:74.0) Gecko/20100101 Firefox/74.0”, followed by the ”Steps to reproduce” or other similar manual text separators. As such, the relevant text to be used further in the research would begin after the first header, ignoring the user agent portion. If no header was found, then the whole text was taken into consideration, otherwise the relevant text was between the the starting headers;

3. Links were removed based on Uniform Resource Locator (URL) composition patterns;

4. **For JIRA issues:** images were removed based on the tags used to insert them in the text.

   For Bugzilla issues: these images are not included in the text but as attachments, so these were not a problem for those BRs;

5. **For JIRA issues:** all of the text in between `{noformat}` and `{code}` tags was removed, as it was commonly used to also input code snippets.

   For Bugzilla issues: code snippets are added in as free-text, so they could not be removed from strings effectively;

6. Number sequences to itemize text (e.g. ”1. ”, ”2. ”, ”3) ” and so on) were removed;

7. Repeated words next to each other due to simple writing mistakes were also removed.

Cleaning Bugzilla issues was a greater challenge compared to JIRA issues, mostly due to the lack of markdown or HTML tags. However, since most of the dataset was composed of JIRA issues, the impact of this challenge is mitigated to a relevant extent. Although this text cleaning process is not perfect, it tries to consider the most cases possible because it is nearly impossible to keep in mind all the possible outliers and different types of combinations users

can make in a non-categorical unstructured input [1]. These actions were created through the observations made on what the most common types of structure (or lack thereof) were whilst in the process of data labeling. Although the length of texts may remain extensive, whether due to imperfections in text cleaning or the complexity of reported bugs, there exists a temptation to employ a keyword extractor (e.g. YAKE!) to potentially streamline data dimensions, in order to reduce computational resource usage and time. However, this approach, while effective for conventional machine learning methods, proves inadequate for BERT-based algorithms. BERT relies heavily on semantic relationships between words for text comprehension, as elucidated in Section 3.6.2. This means that using that approach would likely impair its performance and textual analysis capabilities.

The possibility of employing a paraphraser (i.e. expressing the original meaning in a text using different words) in order to increase variety and dataset size was also explored, through the Parrot Paraphraser, a free tool based on the Transformer architecture [50]. This model was considered due to it being free from cost (as it is opensource), and proven performance in multiple datasets [50]. Soon, the challenge arose of dealing with paraphrasing in unstrucutred text. Most paraphrasing tools, like this one can only work with small to medium size sentences, and not documents. This would be acceptable if this was being used for example on journalistic or literary text, where we are almost sure the sentences are well structured and ending with the correct punctuation. To make the use of this tool possible, a divide and conquer strategy was used, where each space where a newline was defined or big subsets of empty chars were replaced by full stops, and each of these sentences served as input to the Parrot paraphrasing tool, individually. For each sentence, a casing map would also need to be built as the tool's output was all lowercase. With this, the casing could be kept and consistent in words that suffered no changes throughout the paraphrasing. Parrot's `augment` function has multiple parameters that can be fine tuned to improve the paraphrasing qualities that best fit our necessities. Even when using `do_diverse=True` which increases syntactic and phrasal diversity for the output phrases, and reducing both the `adequacy` and `fluency` in order to get a greater variance (sometimes forced due to lack of output), the paraphrases did not vary in a relevant way. After multiple experiments with pre-cleaned bug reports, the results were not helpful on improving the model's performance, actually decreasing it by a lot, mainly due to possible overfitting. This was exacerbated by the max length of the sentences fed as input, which resulted in no outputs or no variation at all compared to the original string, and common outputs where the words were only re-ordered. This could possibly help augment data in data sets with less text complexity, but given the context of this research and the model chosen (and how it processes words - see more in 3.6.2) it did not show results good enough to use in the final implementation.

Another technique commonly used with the same goal as paraphrasing, is back-translation, and it was considered for this study. In this process, sentences are translated into a target language, and re-translated into the original language (in this context, english) to increase text variability. However, applying this technique in this context introduced significant challenges, for example the heightened risk risk of loss of technical accuracy, due to innacuracies in the synonym translation. For example, the term ”cache” in English (denoting a data stor-

age mechanism in computing), may be translated into Portuguese as ”esconderijo”, which translates back to ”hiding place” in English. These discrepancies can lead to a substantial loss of context and meaning of an already highly complex text data. Ensuring the quality of back-translation is an additional layer of manual quality control and analysis that would need to be addressed. Given the constraints of lack of available human resources, conducting thorough checks for each sample would become exceedingly challenging. As a consequence, although this technique can improve text variability, it requires rigorous management to avoid compromising technical precision. Due to these concerns, back-translation was not included in the final implementation of the study. However, it is worth mentioning that this technique has potential value and could be considered for future research.

3.6.2 Model implementation

In these types of implementations, the data fed to the model needs to be of high quality, hence why the text cleaning and filtering is so important, and it is usually a good practice to use a transfer learning approach, which means using a pre-trained model on a similar generalization and refining it to the problem at hand, so that its knowledge can be taken advantage of [51]. Due to the complexity of the textual data, the model chosen for this implementation was BERT, also used in some studies mentioned in the review of the literature, and based on the Transformers architecture [4].

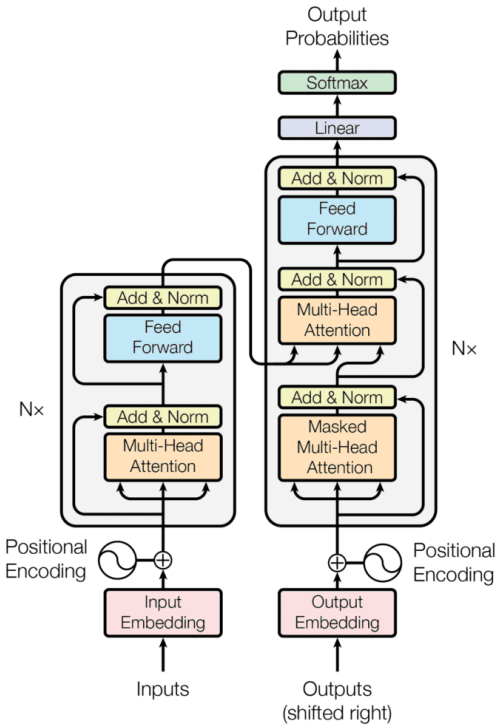


Figure 3.7: The Transformers architecture. The BERT architecture corresponds to the left side of the Figure, relating to the encoder portion [4].

This model shows promising results in various NLP text classification tasks, even with a small amount of labeled data, and since it is pre-trained, it can be fine-tuned to the necessities of

the bug classification task at hand [48]. The process and architecture in pretraining a BERT model is shown in the left diagram of Figure 3.8, and the fine-tuning process is shown on the right side of the same figure, which refers to a question and answer BERT. As observable in that figure, the architectures for both tasks are equal, except for the output layers.

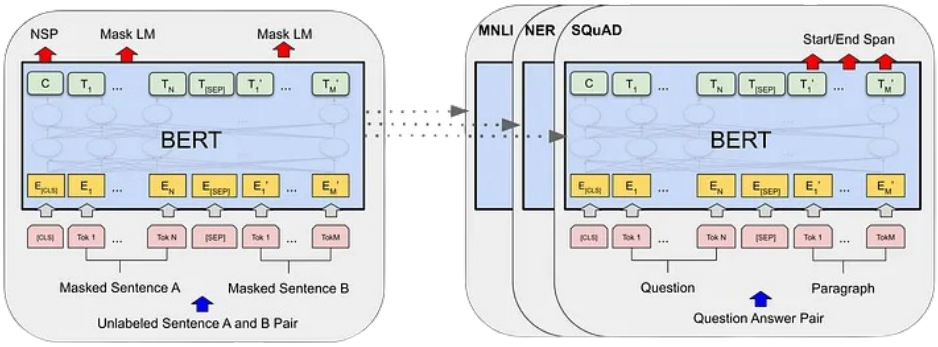


Figure 3.8: BERT architecture through pre-training and fine-tuning procedures [5].

The model uses only the encoder part of the original Transformer model as seen in Figure 3.7, and each encoder layer in BERT is composed by a self-attention mechanism, where just as ChatGPT the importance of each word in relation to the others is weighted followed by a feedforward neural network that refines the token embeddings. To fine-tune BERT in sequence classification, only the last task-specific layer needs to be added on top of the original architecture, which will use one of the special tokens of the encoder to compute the final output (see Section 3.6.2.2).

The researchers who created BERT demonstrated that it achieves state-of-the-art results in tasks such as sentiment analysis, disambiguation of words with various meanings, semantic role labeling, and others [5]. This achievement can be partially explained by how BERT understands and processes words. Instead of taking into account one word at a time (by itself), it relates each word to the whole sentence, which allows the transformer to understand the full context of the text. The model does this process bidirectionally, which means that it takes into account the influence of all surrounding words in both directions, mitigating the inherent bias towards a predetermined meaning that could be introduced if only strictly left-to-right reading was applied [5] [48]. This ability is explained in more detail in Section 3.6.2.2. Furthermore, the disadvantages commonly associated with BERT do not significantly impact the scope of this research, thus justifying its selection for implementation. These drawbacks include [48]:

- **Large model size:** it can pose a threat to researchers who want to deploy the model on less powerful machines with limited computational resources. In this research context, the computational resources are adequate and the model is adjusted so running is possible (see Sections 3.6.2.1 and 3.11);

- **Training time:** can bottleneck researches. Since a pre-trained model and powerful machine were used in this research, this could be somewhat avoided;

- **Language limitation:** since BERT was mostly trained on English data, other applications in other languages may fall short, although researchers have created a pre-trained model trained on 104 languages, and others on individual languages, as Chinese. Since the context of this research and the dataset is in the English language, this limitation had no impact;

- **Model Interpretability:** due to the highly complex architecture involving multiple layers and attention heads, the models nature is deemed ”opaque”. The lack of transparency in the process from the input until a prediction is reached, does not adversely affect the performance and achievements of this research, which plans of leveraging its capabilities instead of interpreting its construction process.

3.6.2.1 BERT model selection

The BERT model is accessible through HuggingFace, and to work with these types of models, the `transformers` library must be used. This library offers a variety of functions and model architectures, including `BertForSequenceClassification`, specifically designed to classify sequences (which could be documents, sentences, or other sequences of words) into certain labels, as the model is followed by a classification head as a layer. There are multiple variations of the BERT model, differing in their training data, i.e. ”bert-base-cased”, ”bert-base-chinese” (and other language variations), ”bert-large-cased-whole-word-masking”, and others. For this study, bert-base-cased was selected due to its widespread use and alignment with the task and the type of data at hand, as well as its performance given the capabilities of the machine on which this implementation was created. It is of utmost importance that the selected BERT model for this implementation is pre-trained on English text, which aligns with the language of our dataset. Furthermore, it is crucial for it to be able to distinguish between cased and uncased words, due to the nature of the dataset. This can be explained by the certainty that some technologies / frameworks / hardware mentioned in the reports are often differentiated with case letters. For example, the pre-trained model has to be able to discern that *python* (the snake) has a different context from *Python* (the programming language). Similarly, if a function named *processUser* is referenced in a report, such as ”error type ’int’ expected but ’string’ found in assignment in function *processUser*”, it is essential that the case-sensitive function name and text integrity are preserved, as these characteristics may significantly impact the model’s ability to accurately classify the BR in the future.

This implementation was tested on a machine provided by Universidade da Beira Interior (UBI) and takes advantage of a 2.2GHz Intel Xeon Silver 4114 processor - 10 total cores with 256 GB RAM, and two NVIDIA Tesla V100 PCIE-16GB Graphics Processing Unit (GPU)s. The provided powerful GPUs played a key role in both the model selection (as they could not handle the largest BERT model bert-large-cased), and in enhancing the performance and training times. The GPUs made it possible to run each epoch in a average of 7 seconds on the smallest models and of 30 seconds on the biggest model, which proved itself to be incredibly useful since there was a necessity to process many epochs, specially in the hyperparameter tuning search across multiple folds (discussed further on Section 3.11).

3.6.2.2 Text Encoding

As previously discussed, the BERT transformer has a great ability to capture representations of language and has proved its efficacy across multiple NLP tasks. However, preparing the data for fine-tuning this model for a downstream task requires some pre-processing steps. With the text already cleaned, the feature data has to be segmented into tokens using the function `BertTokenizer.from_pretrained(bert-base-cased)`, which is based on WordPiece. This algorithm contains a vocabulary with all English characters and the 30.000 most common words and subwords found in the language corpus, designed to create a fixed sized vocabulary for NLP models [52] [6] [53]. WordPiece tokenization breaks text into subwords, helping to handle rare or unseen words of the original vocabulary by breaking them into familiar parts [54]. For instance, the sentence ”The alert dialog box overflows when I browse on mobile.” would be divided into the following subwords ”the”, ”alert”, ”dialog”, ”box”, ”over”, ”##flows”, ”when”, ”I”, ”browse”, ”on”, ”mobile”, assuming that the biggest match for the ”overflows” word is the word ”over”. Another example is shown in Figure 3.9.

Figure 3.9: The WordPiece tokenization process with an example sentence [6].

Subsequently, the BERT Tokenizer adds special tokens to the input sequence, turning into a specific required format, including [53]:

- The `[CLS]` token, marking beginning of the input text;
- The `[SEP]` token, indicating the end of a sentence, or the separation between two sentences.

Each token, both special and string tokens, is then converted into numerical Input IDs. The `[CLS]` and `[SEP]` tokens correspond to the values 101 and 102, respectively. This would convert the previous string into the following Input Id array `[101, 1996, 1188, 12345, 1240, 23253, 2015, 2043, 1045, 17984, 2006, 3186, 1012, 102]`. Additionally, an attention mask is created that consists of an array of ones and zeros. This array indicates which tokens should be attended to and which should be ignored (e.g., padding tokens), a crucial step given the varying lengths of each BR (which averaged around 350 words, title and description included). To ensure uniformity in input lengths, the tokenizer's `truncation` parameter was set to `True` and the `truncation` was set to `max_length`. This ensures that all inputs (future tensors) have the same length, either by shortening or padding them as necessary, with the

special tokens included. As such, the `max_length` is specified as **350**, related to the average string size, although the maximum number of tokens of allowed by bert-base-cased is 512 [5]. Since the implementation uses PyTorch as a Deep Learning framework, the tokenizer parameter `return_tensors` was set as `"pt"` and the input IDs and the attention mask were used to create a `PyTorch TensorDataset`. These `TensorDatasets` were fed into a `DataLoader` with a batch size of 8. The batch size, defined as the number of samples processed before the model is updated, plays a crucial role in determining the efficiency and feasibility of the training. Although larger batch sizes could lead to better results and faster training, the chosen range size needed to be smaller given the GPU limitations. Specifically, a batch size of 32 or 16 would make it impossible to train without running into memory problems, even though both GPUs were being used in parallel. An example of a setup using CUDA to train a model across multiple GPUs can be seen in the Listing 3.2.

Listing 3.2: PyTorch usage with CUDA for parallel processing across both GPUs

```
1 device = torch.device("cuda" if torch.cuda.is_available() else
    "cpu")
2 if torch.cuda.device_count() > 1:
3     modelMainLabel = torch.nn.DataParallel(modelMainLabel)
4 modelMainLabel.to(device)
```

Then, these same `TensorDatasets` were fed into a `DataLoader` with a batch size of 8 due to a limitation of computational capacity, which defines the number of token sequences processed in conjunction in a training iteration (more on 3.6.2.3), and `shuffle` set to `False`.

3.6.2.3 Model Training and Hyperparameter Fine-Tuning

After creating the DataLoaders with the textual data, the training phase could begin. For training and evaluation purposes, K-fold stratified cross validation was used (through Scikit-learn's `StratifiedKfold` method, instead of the usual 80/20 training-test or 80/10/10 training-test-validation split. Stratified cross-validation ensures that the model is trained and evaluated on multiple different subsets of data, where each subset (fold) maintains the same class distribution as the overall dataset, and the same or nearly same sized segments. In this technique, the `K - 1` folds are used for learning, whilst the segment left out is used for validation [55]. This approach provides a more realistic evaluation of the model's performance, particularly crucial in smaller datasets, where maintaining the proportional representation of classes in each fold helps avoid biased and inaccurate performance results. As such, for the main label, which consisted of more than 1000 samples in the dataset, 10 folds were used, as it is the K amount most recommended by researchers [55]. For the subtype models, which had significantly smaller datasets, only 5 folds were defined. Using 10 folds on these smaller subsets (~200 samples) would result in each validation set having too few examples, which could result in lower validation data (which would be an overly optimistic estimate), but would not accurately reflect the true performance of each model, since it would not actually be able to perform true generalization. As such, choosing 5 folds for smaller models, and 10 folds for the bigger one seemed appropriate given the samples available and the balance

achieved between having more data for model training or for validation (bias), providing a reliable estimate of the model's performance, thus reducing the likelihood of underfitting / overfitting.

The training phase also consisted of the usage of the AdamW (an enhanced variant from Adam) optimizer with the `ReduceROnPlateau` scheduler, present in the `transformers` and `PyTorch` libraries, respectively. AdamW is used to adjust each model parameter based on gradient information, in order to get better performance and generalization [56]. The `ReduceROnPlateau` learning rate scheduler monitors the models performance on the validation set, specifically the validation loss. When the scheduler detects that this loss has not improved for a predefined number of epochs (indicating a plateau), it reduces the "global" learning rate that impacts all parameters optimized by AdamW. Both these tools need base parameters to work, such as learning rate and weight decay. These base values were defined in the process of hyperparameter tuning, keeping in mind the ranges in Table 3.11. For the scheduler, the `"mode"` parameter was set to `"min"` which was set to reduce the learning rate when the validation loss reached a minimum and stopped improving; the `"factor"` parameter was set to 0.1 which is the default value for which the learning rate will be multiplied by when a reduction is triggered; and the `"patience"` parameter was set to 2, meaning that if the validation loss does not improve for 2 consecutive epochs, the reduction will be triggered. The base parameters for the learning rate and weight decay required by the AdamW optimizer were determined through hyperparameter tuning, which is detailed below. In addition, early stoppage was incorporated into the training process. This mechanism, implemented manually, stops training if the validation loss does not show improvement over a specified number of epochs (set to 4 in this case), preventing the risk of overfitting and overuse of computational resources that would lead to marginal negligible results.

Regarding hyperparameter selection, the researchers behind BERT noted that while fine-tuning these models is rather simple, the optimal hyperparameters are highly dependent on the task at hand [5]. Nonetheless, the authors provided some possible values that presented positive results in all tasks (such as Next Sentence Prediction, Question Answering, and others) as shown in Table 3.9.

Hyperparameter	Possible Values
Learning rate (Adam)	5e-5, 3e-5, 2e-5
Number of epochs	2, 3, 4

Table 3.9: Range of suggested hyperparameter values to show positive results by Devlin and Chang et al. [5].

These suggestions played a crucial role in defining the range actually used in this implementation. As time complexity was not considered an issue, the selected method of hyperparameter tuning used was Grid Search in the ranges shown in Table 3.10. The grid search technique evaluates the model performance for every combination of hyperparameters in the subset present in the table. It is a very exhaustive "brute-force"-like approach that takes into account the calculated loss in the validation set (which reflects how well the model can generalize on unseen data), as it is a good measure of overfitting.
The hyperparameter ranges chosen were determined with careful consideration of the sug-

Hyperparameter	Possible Values
Learning rate (AdamW)	5e-5, 3e-5, 2e-5, 1e-5
Weight decay	0.1, 0.01, 0.001, 0.0001

Table 3.10: Ranges used hyperparameter tuning for the present implementation.

gestions in Table 3.9, the size of the dataset, the complexity of its text, and the available computational resources and time. Incorporating weight decay as a grid search parameter was crucial to explore its impact on overfitting prevention, which is particularly important given the small size of the dataset. By including both low and high values, we can find the optimal balance between underfitting and overfitting. Even though in the implementation the use of a learning rate scheduler was mentioned, it is still necessary to define the initial starting point for the learning rate hyperparameter. This parameter plays a great role in the performance of the model, which is why it is included in the grid search so the optimal value can be chosen. As for the number of epochs, these were not taken into account in the fine-tuning process because, as mentioned, early stopping was given priority.

The hyperparameter combinations that resulted in the lowest validation losses among the 20 possible configurations were selected by a grid search run for each model. The specific hyperparameters that achieved the minimal validation loss for each model are detailed in Table 3.11.

Model	Input type	Learning rate (Adamw)	Weight Decay	Number of splits (CV)
Main Label	Title	1e-5	0.0001	10
	Description	1e-5	0.0001	10
	Title + Description	1e-5	0.1	10
Security	Title	1e-5	0.1	5
	Description	1e-5	0.01	5
	Title + Description	3e-5	0.0001	5
Code Logic	Title	1e-5	0.01	5
	Description	3e-5	0.001	5
	Title + Description	1e-5	0.1	5
Data	Title	3e-5	0.0001	5
	Description	3e-5	0.1	5
	Title + Description	2e-5	0.1	5
GUI	Title	2e-5	0.1	5
	Description	1e-5	0.01	5
	Title + Description	1e-5	0.1	5

Table 3.11: Hyperparameter combination (except number of splits) with lowest validation loss per model and input, calculated with the Grid Search Technique.

The results presented in Chapter 4 are based on the models trained, fine-tuned, and evaluated using these specific hyperparameter combinations. This thorough approach to hyper-

parameter tuning allowed us to identify the optimal configuration, thereby ensuring that the reported outcomes are reflective of each model's best possible performance.

3.7 Algorithm Evaluation

When evaluating a hierarchical classification algorithm, it is crucial to recognize that each main class has its own dedicated model and, as a consequence, each model should undergo a separate evaluation at their own level within the hierarchy. Although to test the reliability of this proposed solution, it is crucial to show the results of the implementation as a whole, with the same metrics used for individual evaluation.

For the computation and analysis of the metrics delineated in this section, the Scikit-learn library served as the main tool, through its `classification_report` function. The equations pertinent to these computations are described further in this section.

3.7.1 Accuracy

Accuracy is a fundamental metric in machine learning evaluation, especially when the class distribution is balanced, and it is defined as the ratio of correct predictions to the total number of predictions. It is important to keep in mind that a good picture of the model's performance is only obtained when this metric is used in combination with the others mentioned in this subsection, as when used by itself, it can be misleading in unbalanced datasets. This formula is as follows:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}} \quad (3.1)$$

3.7.2 Precision

This metric attempts to measure the proportion of positive identifications that were correct predicted, and is defined as such:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \quad (3.2)$$

Here, TP represents the number of True Positives, and FP represents the number of False Positives. For example, a model that produces no false positives has a precision of 1.0 [57].

3.7.3 Recall

This metric attempts to calculate the proportion of actual positives that was correctly identified, and is defined as such:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \quad (3.3)$$

Here, TP represents the number of True Positives and FN represents the number of False Negatives. For example, a model that does not produce false negatives has a recall of 1.0 [57]. It should be examined along with the precision metric, as generally improving one metric may decrease the other and vice versa. To ensure this, another metric was created called F1-score 3.7.4.

3.7.4 F1-score

This metric offers a balanced assessment that considers both the model's ability to correctly identify positive instances (precision) and its capacity to capture all relevant positive instances (recall) [58]. This metric is particularly advantageous when dealing with unbalanced datasets, where the distribution of classes is uneven. The following formula defines how its calculated:

$$\text{F1-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \quad (3.4)$$

3.8 Conclusion

This chapter provided a comprehensive analysis and detailed description of the tools utilized and the various stages in the construction of the current implementation. Throughout each section, the objective was to enhance the reader's comprehension of the designed implementation by providing justifications for the chosen methodology and decisions made. These include data extraction, preprocessing and labeling; model selection and implementation, fine-tuning choice; evaluation metrics to be employed to showcase performance in the following Results chapter.

Chapter 4

Results

4.1 Introduction

The present chapter provides an analysis and discussion of the performance attained from the previously described implementation, measured using the pertinent metrics outlined in Section 3.7. These results will be juxtaposed with findings from other studies , detailed in Section 2.3, to establish their academic significance.

4.2 Performance evaluation

In order to evaluate the impact of various types of text, it is important to show the various performance results, including the evaluation of the input using only the BRs title, only the description, and both of these inputs in conjunction, as shown in all tables in the following subsections. These results have been measured after hyperparameter tuning.

4.2.1 Main Model

This model takes advantage of the largest sample number (1047), due to its nature, and its performance on multiple labels is presented in Table 4.2. The model's performance varies significantly across the different labels, with the ”Security” and ”Performance” categories demonstrating particularly strong results, which was expected since these are the most direct labels and more restrictive, even though the ”Performance” category is the one with the least number of dataset samples. In the case of the Security label, the model exhibits high precision and F1-score across all input types, indicating a great ability to correctly identify security-related issues. The best performance for this label is observed when using both the title and the description as input, with a precision of 0.86, recall of 0.78, and F1 Score of 0.82. This suggests that the model benefits from a richer input, allowing it to more accurately capture the nuances of security-related bug reports. In contrast, the ”Code Logic” label presents a more moderate performance, which was expected as its the most comprehensive category. Although the F1 score is highest when using both title and description (0.62), the scores are generally lower compared to other labels, except ”Data”. This indicates that the model has some difficulty in accurately classifying these issues, possibly due to the more abstract and varied nature of these reports. The Data label further highlights the model's challenges, particularly with recall. The performance on this label is the weakest versus other categories, and throughout the various input types. This shows that the model struggles to correctly identify these reports and has a high number of false negatives. The performance metrics for this label are lower overall. and the results on this label fall short, so this requires ex-

tra attention as it could hinder the model's effectiveness and usability in practical applications where accurate classification of Data-related reports is essential. For the GUI label, the model maintains moderate performance across all input types. The precision and recall are fairly balanced, resulting in a stable F1-score around 0.74-0.76 depending on the input. This consistency indicates that the model has a reasonable understanding of GUI-related issues. The ”Performance” label stands out with strong precision and F1-scores, particularly when using both title and description as inputs. The model achieves a precision of 0.86 and an F1 Score of 0.85 with this combined input type, reflecting its very positive ability to accurately classify performance-related issues, despite the lower number of samples. This high level of performance is crucial to ensure that critical performance bugs are identified and addressed promptly.

Main Type	Input Type	Precision	Recall	F1 Score
Security	Title	0.81	0.77	0.79
	Description	0.77	0.65	0.71
	Title + Description	0.86	0.78	0.82
Code Logic	Title	0.59	0.61	0.60
	Description	0.54	0.6	0.57
	Title + Description	0.61	0.64	0.62
Data	Title	0.52	0.57	0.54
	Description	0.44	0.51	0.47
	Title + Description	0.58	0.61	0.59
GUI	Title	0.75	0.74	0.74
	Description	0.67	0.68	0.68
	Title + Description	0.76	0.75	0.76
Performance	Title	0.85	0.8	0.82
	Description	0.81	0.66	0.73
	Title + Description	0.86	0.85	0.85

Table 4.1: Performance evaluation of the Main Label model, per each label and input type, with hyperparameter optimization.

Input Type	Accuracy
Title	0.69
Description	0.62
Title + Description	0.72

Table 4.2: Accuracy distribution of the Main label model per input type.

Overall, this evaluation reveals several strengths. The model performs exceptionally well in identifying ”Security” and ”Performance” bug reports, making it a valuable tool in identifying likely critical security bug reports, and showing that the usage of the title and the description yields better results. While the model shows some challenges with the ”Code Logic” and ”Data” labels, it is important to recognize that the model still maintains a usable level of performance in these categories. Specifically, in the context of real-world software development, these F1-scores of 0.62 and 0.59 (respectively) suggest that the model can still be effectively employed to assist in bug classification processes for ”Code Logic” and ”Data” issues, although it might need some supervision. Developers can partially rely on the model to provide meaningful classifications, even if some manual verification may be needed for the most complex or ambiguous cases.

4.2.2 Security subtypes model

This model takes advantage of 203 samples. Despite the small dataset, the performance evaluation shown in Table 4.4 provides some key information on the effectiveness of the model among different subtypes of security bug reports. For the ”Authorization” category, the model has a very robust performance, yielding respectable F1-scores of 0.81, 0.62, and 0.77 when both the BR’s title and Description are used. However, for the ”System misconfigurations” subtype, which comprises a smaller subset of more complex and challenging data in comparison to the other subtypes, presented an extremely low recall when only either the title or the description were used, but an acceptable result when these were concatenated and used as input. This category benefitted heavily from the full context that using both inputs provides textually, since the margin between results is very noticeable.

Security Subtype	Input Type	Precision	Recall	F1 Score
Authorization	Title	0.71	0.78	0.74
	Description	0.75	0.75	0.75
	Title + Description	0.82	0.80	0.81
System Misconfigurations	Title	0.56	0.36	0.43
	Description	0.43	0.31	0.36
	Title + Description	0.64	0.6	0.62
Vulnerabilities	Title	0.7	0.76	0.73
	Description	0.66	0.75	0.7
	Title + Description	0.74	0.79	0.77

Table 4.3: Performance evaluation of the Security subtypes model, per each label and input type, with hyperparameter optimization.

Input Type	Accuracy
Title	0.69
Description	0.66
Title + Description	0.75

Table 4.4: Accuracy distribution of the Security subtypes model per input type.

As observable in Table 4.4, the accuracy results also reflect the findings mentioned previously, since the highest accuracy is achieved with title and description as input, showing that the model performs well overall, despite struggling with one of the subtypes. The elevated accuracy observed in the model despite the small number of samples can be attributed, in part, to the relatively limited number of labels available within this security subtype. This scarcity of labels can artificially inflate accuracy metrics. However, this result also highlights the efficacy of the hierarchical classification approach used in this study, which appears to effectively leverage the available data to enhance accuracy.

4.2.3 Code Logic subtypes model

This model takes advantage of 232 samples and also has the most comprehensive subtypes, so some underperformance could be predicted due to the higher variance of the data and a higher number of available labels. For most subtypes, the use of title and description as input generally leads to the best performance, as indicated by higher F1 scores and accuracy as seen in Table 4.5. Specifically, the ”Internal Search Engine” subtype achieves the highest F1 score of 0.84 with combined inputs, reflecting a strong balance between precision (0.89) and recall (0.80). In contrast, the ”Functional” subtype underperforms across all metrics, with the highest F1 score being only 0.57, suggesting difficulties in accurately iden-

tifying functional logic, mainly due to how variable this class is since it is the most subjective one. The accuracy distribution follows a similar trend, with the highest accuracy (0.70) achieved when using both Title and Description, indicating that combining these inputs generally leads to a higher model reliability.

Code Logic Subtype	Input Type	Precision	Recall	F1 Score
Calculation	Title	0.71	0.72	0.72
	Description	0.82	0.68	0.74
	Title + Description	0.77	0.79	0.78
Functional	Title	0.5	0.63	0.56
	Description	0.47	0.55	0.51
	Title + Description	0.57	0.57	0.57
Internal Search Engine	Title	0.9	0.72	0.8
	Description	0.68	0.78	0.73
	Title + Description	0.89	0.8	0.84
Libraries	Title	0.73	0.69	0.71
	Description	0.6	0.62	0.61
	Title + Description	0.62	0.77	0.69
API Integration	Title	0.6	0.58	0.59
	Description	0.68	0.56	0.61
	Title + Description	0.68	0.6	0.64

Table 4.5: Performance evaluation of the Code Logic subtypes model, per each label and input type, with hyperparameter optimization.

Input Type	Accuracy
Title	0.66
Description	0.55
Title + Description	0.70

Table 4.6: Accuracy distribution of the Code Logic subtypes model per input type.

However, the performance variation across subtypes highlights that the model's effectiveness is context-dependent, excelling in some areas (like "Internal Search Engine") while struggling in others (such as "Functional"). These results indicate that, while the model demonstrates robustness for certain logic subtypes, further refinement is required for those categories with lower performance metrics. It is likely that these underperforming subtypes would benefit from an increased sample size, allowing the model to incorporate a wider range of scenarios and contexts. This could enhance model performance in these categories, which, unlike the better-performing classes, do not appear to follow a consistent pattern. Despite the need for further refinement in certain subtypes, the model still provides a helpful and insightful picture when both Title and Description inputs are utilized, as it performs with an accuracy of 70%.

4.2.4 Data subtypes model

The subset of the data on which this model was trained consisted of 200 samples, and its performance across subtypes can be seen in Tables 4.7 and 4.8. These numbers show that the lowest performance occurs in the category "Incorrect Data Structure", with an F1 score below 0.5, even when the title and description are combined as input. for this label, the model really struggles with this label, despite the huge margin of and performance improvement across the various input types. The "Data Transmission" subtype shows moderate performance; however, the moderate F1 scores indicate that the model may still struggle with

certain aspects of data transmission problems, despite being the ”Data” subtype with most of the labeled samples.

Data Subtype	Input Type	Precision	Recall	F1 Score
Incorrect Data Structure	Title	0.26	0.3	0.28
	Description	0.28	0.19	0.22
	Title + Description	0.48	0.44	0.46
Memory Faults	Title	0.77	0.83	0.8
	Description	0.85	0.61	0.71
	Title + Description	0.96	0.75	0.84
Pointer Faults	Title	0.94	0.85	0.89
	Description	0.53	0.64	0.58
	Title + Description	0.84	0.97	0.9
Data Transmission	Title	0.51	0.44	0.47
	Description	0.5	0.58	0.54
	Title + Description	0.62	0.6	0.61
Encoding	Title	0.75	0.81	0.78
	Description	0.67	0.71	0.69
	Title + Description	0.76	0.85	0.8

Table 4.7: Performance evaluation of the Data subtypes model, per each label and input type, with hyperparameter optimization.

Input Type	Accuracy
Title	0.66
Description	0.58
Title + Description	0.74

Table 4.8: Accuracy distribution of the Data subtypes model per input type.

In contrast, the model performs exceptionally well in identifying ”Memory Faults” and ”Pointer Faults”, with F1 Scores reaching 0.84 and 0.9, respectively, which was expected since these BRs tend to be described in very similar ways, commonly including null pointer exception and other similar logs in their report. This consistency helps BERT recognize the traces in these categories, resulting in better precision and recall performance. Finally, the ”Encoding” subtype demonstrates strong performance across multiple input types but shows the best performance when both are used, resulting in an F1 score of 0.80. This suggests that encoding issues are well represented in the dataset, and the model is proficient at identifying them when given sufficient contextual information.

4.2.5 GUI subtypes model

This model takes advantage of 280 samples, the biggest subset of data available for a subtype on the dataset, and several insights can be drawn regarding its effectiveness through the results in Tables 4.9 and 4.10. This model is the only one that performs slightly worse when given more textual context, as the results in accuracy and other metrics tend to have a larger margin when using only the BR titles as input versus combining both inputs, breaking the pattern observed in other models so far. The results are also surprisingly low, despite the larger sample number. Although with an average precision of 0.58, an average recall of 0.54 and an average F1 score of 0.55, the model can reasonably precise in its predictions, despite occasionally wrongly identifying relevant instances, suggesting that further optimization or additional feature engineering could help in capturing a broader range of relevant data across the different available subtypes.

GUI Subtype	Input Type	Precision	Recall	F1 Score
Navigation	Title	0.5	0.45	0.47
	Description	0.4	0.34	0.37
	Title + Description	0.48	0.61	0.54
Style	Title	0.59	0.43	0.49
	Description	0.31	0.36	0.33
	Title + Description	0.67	0.46	0.55
Validations	Title	0.6	0.69	0.64
	Description	0.55	0.46	0.5
	Title + Description	0.61	0.63	0.62
Content	Title	0.45	0.53	0.48
	Description	0.41	0.49	0.45
	Title + Description	0.46	0.56	0.5
Accessibility	Title	0.7	0.7	0.7
	Description	0.55	0.5	0.52
	Title + Description	0.69	0.45	0.55

Table 4.9: Performance evaluation of the GUI subtypes model, per each label and input type, with hyperparameter optimization.

Input Type	Accuracy
Title	0.55
Description	0.42
Title + Description	0.546

Table 4.10: Accuracy distribution of the GUI subtypes model per input type.

As such, these tables show that while the model demonstrates promising performance in classifying GUI subtypes, particularly with optimal hyperparameters, there is still a lot of room for improvement and refining in this model.

4.3 Result discussion

The overall results presented show the importance of leveraging the full textual content of bug reports to improve the classification accuracy. The vast majority of models showed enhanced performance using both the title and the description as inputs, although the results of using only the title of the bug report were surprisingly positive, even more so considering the small lengths of this type of text. The surprising positive results with titles can be partially explained by the necessity of describing the observed issue using as many keywords as possible, in a smaller format, leading to less cluttered text, written in a more objective way, which is positive. The contrary can be said as to why using only the descriptions yields the worst performances overall, resulting in a decrease in accuracy of almost 17% in average when compared to title and description combined, versus the 4.5% decrease in title-only inputs. Even though this text was ”cleaned” and pre-processed, this technique is nearly impossible to perfect because non-categorical text is highly variable. This, associated with BERTs limitation on processing long sentences, can explain the poorer results on description-only bug reports as input. This aspect of the performance partially co-relates with the findings in [26] related to the positive impact on performance of the bug description, despite the amount a lot of noisy text, although in the present findings this is only true when its concatenated with the title.

In contrast to the studies referenced in Section 2.3, this research was carried out using a relatively smaller dataset, especially when compared to such as Santos et al. [40] and Hem-

mati et al. [41], that used datasets comprising of 700 000 and 3 000 samples (versus the current 1 047 samples). Despite the substantial differences in dataset size, the performance discrepancy – particularly in the case of Santos et al. [40], who used exactly the same types of inputs as the present study – remains relatively minor. Regardless of this limitation, several models within the present study, particularly those associated with the ”Security” subtype and the Main type, exhibited strong overall performance, achieving peak accuracies of 75% and 72%, respectively. This suggests that leveraging predictions from a model trained in a more generalized context and subsequently refining these predictions into more specific and informative subtypes can be beneficial, even if the results do not fully match the performance reported in studies such as those by Catolino et al. [7], Kallis et al. [36], or Persing et al. [31]. Although these studies achieved favorable results, it is important to note that they dealt with a smaller number of possible labels (averaging 5 categories) compared to the 18 possible type combinations in this study, which offer a more detailed characterization of bug reports to whomever might be responsible for fixing the bug. Consequently, the approach taken in this study, while potentially requiring additional refinement or validation, demonstrates the potential of extracting rich information from more complex classification tasks.

These findings highlight the inherent robustness of the BERT / Transformer architectures, particularly when applied to tasks involving smaller datasets. This model has proven once again its capability of delivering competitive and promising outcomes, even in scenarios where data availability is limited, either due to its nature or due to the complexity involving a manual labeling process. This success mainly relies on the transfer learning and fine-tuning techniques through which the model can leverage pre-existing classification knowledge, adapting itself to new specific tasks, with high versatility. Despite the promising results, some of the subtype models can be substantially improved and would likely achieve even better performances across all labels if the dataset comprised of more samples. This could be accomplished by incorporating additional manual labeling with diverse textual contexts, utilizing a hybrid approach that combines supervised learning with reinforcement learning techniques, or improving the implemented text preprocessing methods.

4.4 Conclusion

This chapter presented the performance evaluations of each model using the metrics outlined in the Methodology Section and provided conclusions based on these evaluations, offering a justification for the results achieved. Furthermore, the results were compared with the most relevant findings discussed in the Review of the literature (Section 3.6.2) to contextualize and validate the performance of the study in relation to existing work.

Chapter 5

Main conclusions and Future Work

5.1 Introduction

The final chapter of this dissertation presents the main conclusions of the research carried out, highlighting its relevance to the field and how it can impact future studies and real-world scenarios. In addition, potential future enhancements to improve the implementation developed in this study are discussed.

5.2 Main conclusions

At the time of this investigation, it was possible to observe that the field of bug classification has been studied since the concept of software was created. Development teams have always felt a need to assign labels to bugs to study their product, business, the quality of the software being built, and also just for organization purposes. Many bugs can have catastrophic results on product quality and end-user trust, especially those related to security and functionality, and, as such, identifying and fixing these as quickly as possible can be a huge portion of a software company's success. With the evolution of how software is being built, new layers of complexity have started to be added to classification schemes, although most of those so far seem to be based on the same constant foundations. Despite this evolution, the most recent schemes seem to be too broad, incorporating multiple similar bugs within the same classifications, mostly because software is very intertwined. When dealing with bug report classification, this research provides future researchers with a more comprehensive and detailed classification scheme, which could be used in real-word scenarios and, as a result, gives developers an inner sense of what the intentions of the bug report are, even when these descriptions are made by end-users that lack software knowledge. Splitting the classification categories into a divide-and-conquer problem allowed the possibility to split a small dataset into a larger number of models. This means that through this technique, a larger number of more detailed labels can be assigned to each bug report.

Traditionally, software teams rely on the entire bug report analysis process on a human being daily, which increasingly tends to result in errors due to repetitiveness, and although it pays off, the cost of these operations could be decreased with the help of automation. With automatic classification of these issues, the time investment could go toward bug-fixing processes or other software development options instead, leading to higher software quality and smaller bug fix times. Despite the need for further refinement in certain classes, the implementation demonstrated overall success, especially considering the conditions, the complexity, and the context of the data available. The performance evaluation showed that the use of BERT is a reliable, robust, and high performing option to develop tools that improve bug

triage. It also provides a more streamlined preprocessing method compared to traditional NLP techniques such as TF-IDF, dimensionality reduction, and others. The way this model and its architecture can use small datasets and its transfer learning techniques to its advantage were proven, showing that this Transformer should be used in similar contexts. In spite of the fact that the results on some of the models could be improved, in a short-term, the implementation could be enhanced with some verification from the developers assigned to the bug report analysis, giving their feedback to the model, but in the long term, building on this approach could lead to a fully automated bug classification process. Separation of the classification scheme into various types and subtypes could enhance memorability, particularly if integrated into the said feedback mechanisms.

5.3 Future Work

During the development and testing of this implementation, several improvements were identified that would make this classification tool even more useful for Quality Assurance teams, and Software Quality / NLP researchers were identified. For example, we should strive for better bug tracking systems or bug description platforms. Although allowing common users with lack of software knowledge to freely write about their issues can seem like the most optimal solution, sometimes less is more. Some limitations would probably lead to more objective descriptions, which can be extended /(if needed) in the already existing comment sections. Giving users different input fields for different types of information, thus indirectly ”forcing” them to structure their report, would also help researchers and data miners extract exactly what they need for their research. A less ”extreme” solution would be to provide users with a script of ”To-Do’s” or ”Don’t Do’s” to be followed and only allow bug reports that comply with it, to a certain extent, as some flexibility might be needed. As a result, interesting parties could then trust that their data fields would not be as cluttered and noise-filled, which would likely lead to better research in this field. Creating a hybrid of this supervised tool with some reinforced learning where the developers of a specific project could tune the implementation even more, could also likely lead to the best performance in a long-term real-life scenario, although none of the studies mentioned seem to have tried this possibility. Finally, employing this tool as a type of open-source plugin for a BTS like GitHub so it can be used in conjunction with similar classifiers (e.g. bug priority classifiers) on a large number of contexts is also an interesting future feature.

Bibliography

[1] C. E. Öztürk and O. Köksal, "A survey on machine learning-based automated software bug report classification," in *2022 International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)*, 2022, pp. 635–640. xiii, 2, 17, 34

[2] T. Sultan, A. Khedr, and M. Sayed, "A proposed defect tracking model for classifying the inserted defect reports to enhance software quality control," *Acta informatica medica : AIM : journal of the Society for Medical Informatics of Bosnia and Herzegovina : časopis Društva za medicinsku informatiku BiH*, vol. 21, pp. 103–8, 04 2013. xiii, 8

[3] R. Grady B., *Software Failure Analysis for High-Return Process Improvement Decisions - Hewlett-Packard Journal*. Hewlett-Packard Co., 1996. xiii, 1, 7, 8

[4] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 6000–6010. xiii, 35

[5] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *North American Chapter of the Association for Computational Linguistics*, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:52967399 xiii, xv, 16, 36, 39, 40

[6] Google Research, "A fast wordpiece tokenization system." [Online]. Available: http://research.google/blog/a-fast-wordpiece-tokenization-system/ xiii, 38

[7] G. Catolino, F. Palomba, A. Zaidman, and F. Ferrucci, "Not all bugs are the same: Understanding, characterizing, and classifying bug types," *Journal of Systems and Software*, vol. 152, pp. 165–181, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121219300536 xv, 5, 6, 14, 18, 23, 28, 51

[8] IEEE, "IEEE standard glossary of software engineering terminology," *IEEE Std 610.12-1990*, pp. 1–84, 1990. 1

[9] S. C. McConnell, *Code Complete - A Practical Handbook Of Software Costruction*. Microsoft Press, 2004. 1

[10] R. B. Grady and D. L. Caswell, *Software Metrics: Establishing a Company-Wide Program*. Prentice Hall, 1987. 1

[11] S. Bhattacharya, S. Radha, and D. Jat, "Comparative analysis of bug tracking tools," *International Journal of Pharmacy and Technology*, vol. 8, pp. 4989–4998, 12 2016. 1

[12] M. Gegick, P. Rotella, and T. Xie, "Identifying security bug reports via text mining: An industrial case study," pp. 11 – 20, 06 2010. 2

[13] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: How misclassification impacts bug prediction," Universität des Saarlandes, Saarbrücken, Germany, Tech. Rep., August 2012. 2, 19, 23

[14] L. Jonsson, M. Borg, D. Broman, K. Sandahl, S. Eldh, and P. Runeson, "Automated bug assignment: Ensemble-based machine learning in large scale industrial contexts," *Empirical Software Engineering*, vol. 21, 09 2015. 2

[15] B. Freimut, C. Denger, and M. Ketterer, "An industrial case study of implementing and validating defect classification for process improvement and quality management," *Proceedings - International Software Metrics Symposium*, vol. 2005, pp. 10 pp.–, 10 2005. 5, 22

[16] R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. Ray, and M.-Y. Wong, "Orthogonal defect classification - a concept for in-process measurements," *Software Engineering, IEEE Transactions on*, vol. 18, pp. 943 – 956, 12 1992. 5, 6, 9, 12

[17] I. Lopes Margarido, J. P. Faria, R. M. Vidal, and M. Vieira, "Classification of defect types in requirements specifications: Literature review, proposal and assessment," pp. 1–6, 2011. 5

[18] L. Yu, C. Kong, L. Xu, J. Zhao, and H. Zhang, "Mining bug classifier and debug strategy association rules for web-based applications," pp. 427–434, 10 2008. 7, 11, 18

[19] A. Endres, "An analysis of errors and their causes in system programs," *IEEE Transactions on Software Engineering*, vol. SE-1, no. 2, pp. 140–149, 1975. 8

[20] T. J. Ostrand and E. J. Weyuker, "Collecting and categorizing software error data in an industrial environment," *J. Syst. Softw.*, vol. 4, pp. 289–300, 1982. [Online]. Available: https://api.semanticscholar.org/CorpusID:36415815 9

[21] C. B. Seaman, F. Shull, M. Regardie, D. Elbert, R. L. Feldmann, Y. Guo, and S. Godfrey, "Defect categorization: Making use of a decade of widely varying historical data," New York, NY, USA, p. 149–157, 2008. [Online]. Available: https://doi.org/10.1145/1414004.1414030 9

[22] M. Sullivan and R. Chillarege, "A comparison of software defects in database management systems and operating systems," pp. 475–484, 1992. 10

[23] T. Sultan, A. Khedr, and M. Sayed, "A proposed defect tracking model for classifying the inserted defect reports to enhance software quality control," *Acta informatica medica : AIM : journal of the Society for Medical Informatics of Bosnia and Herzegovina : časopis Društva za medicinsku informatiku BiH*, vol. 21, pp. 103–8, 04 2013. 10

[24] "History of GUIs," 2024. [Online]. Available: https://you.stonybrook.edu/historyofguis/2024/01/10/history-of-guis/ 10

[25] N. Neelofar, M. Javed, and H. Mohsin, "An automated approach for software bug classification," pp. 414–419, 07 2012. 11

[26] S. K. Kumarasamy Mani, A. Sankaran, and R. Aralikatte, "Deeptriage: Exploring the effectiveness of deep learning for bug triaging," *CoDS-COMAD '19: Proceedings of the ACM India Joint International Conference on Data Science and Management of Data*, pp. 171–179, 2019. 11, 12, 50

[27] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, "Bug characteristics in open source software," *Empirical Softw. Engg.*, vol. 19, no. 6, p. 1665–1705, dec 2014. [Online]. Available: https://doi.org/10.1007/s10664-013-9258-8 12, 18

[28] F. Thung, X.-B. D. Le, and D. Lo, "Active semi-supervised defect categorization," in *2015 IEEE 23rd International Conference on Program Comprehension*, 2015, pp. 60–70. 12, 18

[29] F. Thung, D. Lo, and L. Jiang, "Automatic defect categorization," in *2012 19th Working Conference on Reverse Engineering*, 2012, pp. 205–214. 12, 18, 23

[30] T. Joachims, "SVMmulticlass," Aug 2008. [Online]. Available: https://www.cs.cornell.edu/people/tj/svm_light/svm_multiclass.html 12

[31] L. Huang, V. Ng, I. Persing, M. Chen, Z. Li, R. Geng, and J. Tian, "AutoODC: Automated generation of orthogonal defect classifications." *Automated Software Engineering*, vol. 22, pp. 3–46, 03 2015. 12, 13, 18, 51

[32] F. Lopes, J. Agnelo, C. A. Teixeira, N. Laranjeiro, and J. Bernardino, "Automating orthogonal defect classification using machine learning algorithms," *Future Generation Computer Systems*, vol. 102, pp. 932–947, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X19308283 13

[33] N. K. Nagwani and S. Verma, "Software bug classification using suffix tree clustering (stc) algorithm," 2011. [Online]. Available: https://api.semanticscholar.org/CorpusID:16906001 13

[34] M. Castellucio and S. Ledru, "Teaching machines to triage firefox bug," 4 2019. [Online]. Available: https://hacks.mozilla.org/2019/04/teaching-machines-to-triage-firefox-bugs 14, 18

[35] M. Castelluccio, "bugbug," Aug. 2021. [Online]. Available: https://github.com/mozilla/bugbug 14

[36] R. Kallis, A. Di Sorbo, G. Canfora, and S. Panichella, "Ticket tagger: Machine learning driven issue classification," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 406–409. 14, 18, 51

[37] K. Kowsari, K. Jafari Meimandi, M. Heidarysafa, S. Mendu, L. Barnes, and D. Brown, "Text classification algorithms: A survey," *Information*, vol. 10, no. 4, 2019. [Online]. Available: https://www.mdpi.com/2078-2489/10/4/150 15

[38] OpenAI, "GPT-4 - Research," 2024. [Online]. Available: https://openai.com/gpt-4 16

[39] A. Kumar, "Analytics yogi - large language models (LLM): Types, examples, use cases," 2023. [Online]. Available: https://vitalflux.com/large-language-models-concepts-examples/ 16

[40] M. Siddiq and J. S. Santos, "BERT-based GitHub issue report classification," in *2022 IEEE/ACM 1st International Workshop on Natural Language-Based Software Engineering (NLBSE)*, vol. 1. Los Alamitos, CA, USA: IEEE Computer Society, may 2022, pp. 33–36. [Online]. Available: https://doi.ieeecomputersociety.org/ 16, 18, 50, 51

[41] E. Mashhadi, H. Ahmadvand, and H. Hemmati, "Method-level bug severity prediction using source code metrics and LLMs," 09 2023. 16, 18, 51

[42] K. Alrashedy, "Language models are better bug detector through code-pair classification," 2023. 16

[43] Python Software Foundation, "What is python? executive summary." [Online]. Available: https://www.python.org/doc/essays/blurb/?external_link=true 19

[44] K. R. Srinath, "Python – the fastest growing programming language," 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:218773526 19

[45] Scikit-learn, "Scikit-learn - about us," 2024. [Online]. Available: https://scikit-learn.org/stable/about.html 20

[46] "Bugzilla documentation." [Online]. Available: https://bugzilla.readthedocs.io/en/5.0.4/api/core/v1/bug.html 21

[47] "The bugzilla guide - classifications." [Online]. Available: https://www.bugzilla.org/docs/2.20/html/classifications.html 21

[48] P. Ray, "Chatgpt: A comprehensive review on background, applications, key challenges, bias, ethics, limitations and future scope," *Internet of Things and Cyber-Physical Systems*, vol. 3, 04 2023. 25, 36

[49] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020. [Online]. Available: https://arxiv.org/abs/2005.14165 25

[50] P. Damodaran, "Parrot: Paraphrase generation for NLU." 2021. 34

[51] AltexSoft, "Semi-supervised learning, explained." [Online]. Available: https://www.altexsoft.com/blog/semi-supervised-learning 35

[52] HuggingFace, "Huggingface transformers - tokenizers." [Online]. Available: https://huggingface.co/docs/transformers/v4.42.0/en/model_doc/bert#transformers.BertTokenizer 38

[53] C. McCormick, "BERT word embeddings," 2019. [Online]. Available: https://mccormickml.com/2019/05/14/BERT-word-embeddings-tutorial/#2-input-formatting 38

[54] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, Łukasz Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean, "Google's neural machine translation system: Bridging the gap between human and machine translation," 2016. [Online]. Available: https://arxiv.org/abs/1609.08144 38

[55] P. Refaeilzadeh, L. Tang, and H. Liu, *Cross-Validation*. Boston, MA: Springer US, 2009, pp. 532–538. [Online]. Available: https://doi.org/10.1007/978-0-387-39940-9_565 39

[56] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," 2019. [Online]. Available: https://arxiv.org/abs/1711.05101 40

[57] Google, "Classification: Precision and Recall | Machine Learning - Google for Developers," 2022. [Online]. Available: https://developers.google.com/machine-learning/crash-course/classification/precision-and-recall. 42, 43

[58] S. Adhikarla, "Automated bug classification: Bug report routing," 2020. 43