UNIVERSIDADE FEDERAL DO CEARÁ

CENTRO DE CIÊNCIAS

DEPARTAMENTO DE COMPUTAÇÃO

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ANTÔNIO JOSÉ AMÂNCIO DA SILVA

TOWARDS AUTOMATIC LABELING OF EXCEPTION HANDLING BUGS: A CASE STUDY OF 10 YEARS BUG-FIXING IN APACHE HADOOP

FORTALEZA

2022

ANTÔNIO JOSÉ AMÂNCIO DA SILVA

TOWARDS AUTOMATIC LABELING OF EXCEPTION HANDLING BUGS: A CASE STUDY OF 10 YEARS BUG-FIXING IN APACHE HADOOP

Dissertação apresentada ao Curso de do Programa de Pós-Graduação em Ciência da Computação do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do título de mestre em Ciência da Computação. Área de Concentração: Engenharia de Software

Orientador: Prof. Dr. Lincoln Souza Rocha

FORTALEZA

2022

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

S578t Silva, Antônio José Amâncio da.
Towards automatic labeling of exception handling bugs: a case study of 10 years bug-fixing in apache hadoop / Antônio José Amâncio da Silva. – 2022.
53 f. : il. color.

Dissertação (mestrado) – Universidade Federal do Ceará, Centro de Ciências, Programa de Pós-Graduação em Ciência da Computação, Fortaleza, 2022.
Orientação: Prof. Dr. Lincoln Souza Rocha.

1. Exception Handling Bug. 2. Automatic Bug Labeling. 3. Machine Learning. 4. Natural Language Processing. I. Título.

CDD 005

ANTÔNIO JOSÉ AMÂNCIO DA SILVA

TOWARDS AUTOMATIC LABELING OF EXCEPTION HANDLING BUGS: A CASE STUDY OF 10 YEARS BUG-FIXING IN APACHE HADOOP

Dissertação apresentada ao Curso de do Programa de Pós-Graduação em Ciência da Computação do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do título de mestre em Ciência da Computação. Área de Concentração: Engenharia de Software

Aprovada em: 25 de Novembro de 2022

BANCA EXAMINADORA

Prof. Dr. Lincoln Souza Rocha (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Paulo Henrique Mendes Maia
Universidade Estadual do Ceará (UECE)

Prof. Dr. João Paulo Pordeus Gomes
Universidade Federal do Ceará (UFC)

Dedico esta dissertação aos meus pais e à minha esposa, que sempre me apoiaram. Um agradecimento especial à minha mãe, que me ensinou a importância do estudo. Essa conquista é fruto do incentivo que recebi de vocês.

ACKNOWLEDGEMENTS

A Deus, por me conceder saúde, força e sabedoria para enfrentar todos os desafios ao longo dessa jornada. Sem a fé e a esperança que Ele me proporcionou, esse sonho não teria se concretizado.

À minha família, que sempre foi meu alicerce. À minha mãe, Verônica Amâncio, e ao meu pai, Francisco José Gildo, que com amor, paciência e apoio incondicional me guiaram em todos os momentos dessa caminhada. À minha esposa, Renata Carvalho, por ser minha companheira, me encorajar e acreditar em mim, mesmo nos dias mais difíceis.

Aos colegas dos laboratórios de pesquisa GREat, Arida e LSBD, que foram parte essencial dessa jornada. A cada um de vocês, meu muito obrigado pelo companheirismo, trocas de ideias e por todos os momentos compartilhados durante essa fase.

Ao meu orientador, Prof. Lincoln Souza Rocha, pela orientação precisa, paciência e por sempre acreditar no potencial deste trabalho. Ao Professor João Paulo P. Gomes, pelos ensinamentos e contribuições valiosas.

Aos colegas Diego Mesquita, Renan Vieira e Juarez Meneses Filho, que me ajudaram de forma significativa na conclusão desse trabalho. Sua colaboração e amizade foram fundamentais para que eu pudesse chegar até aqui.

A todos que, de alguma forma, fizeram parte dessa jornada, deixo meu sincero agradecimento.

"Nunca desista de um sonho por causa do tempo que levará para realizá-lo. O tempo vai passar de qualquer maneira."

(Earl Nightingale)

RESUMO

Contexto: Os bugs de tratamento de exceções (EH) decorrem do uso incorreto do mecanismo de tratamento de exceção (EHM) e frequentemente acarretam consequências severas (e.g., tempo de inatividade do sistema, perda de dados e risco de segurança). O rastreamento de bugs EH é particularmente relevante para sistemas contemporâneos (como sistemas baseados em nuvem e inteligência artificial), nos quais a lógica sofisticada do software representa uma ameaça adicional ao uso correto do EHM. Além disso, as pessoas que reportam bugs raramente conseguem rotular bugs como bugs EH, pois isso pode exigir um conhecimento abrangente da estratégia de EH do software. Surpreendentemente, até onde sabemos, não existe um procedimento automatizado para identificar bugs EH a partir das descrições dos relatórios. Objetivo: Primeiramente, buscamos avaliar até que ponto o Processamento de Linguagem Natural (NLP) e o Aprendizado de Máquina (ML) podem ser usados para rotular de forma confiável os bugs EH utilizando os campos de texto dos relatórios de bugs (e.g., resumo, descrição e comentários). Em segundo lugar, pretendemos fornecer um conjunto de dados rotulados de maneira confiável que a comunidade possa usar em esforços futuros. De modo geral, esperamos que nosso trabalho aumente a conscientização da comunidade sobre a importância dos bugs EH. Método: Analisamos manualmente 4.516 relatórios de bugs dos quatro principais componentes do projeto Hadoop da Apache, dos quais rotulamos cerca de $\approx 20\%$ (943) como bugs EH. Em seguida, utilizamos técnicas de incorporação (*embedding*) de palavras (Bag-of-Words e Frequência de Termos - Frequência Inversa de Documentos – TF-IDF) para resumir os campos textuais dos relatórios de bugs. Posteriormente, usamos essas incorporações para ajustar quatro classes de métodos de ML e registrar seu desempenho em dados não vistos. Também avaliamos se a consideração exclusiva de palavras-chave de EH é suficiente para alcançar um alto desempenho preditivo. Resultados: Nossos resultados mostram que a combinação de técnicas de NLP e ML pode rotular bugs EH de forma razoavelmente eficaz, alcançando pontuações de Características de Operação do Receptor - Área Sob a Curva (ROC-AUC) de até 0,70 e recall variando de 0,50 a 0,62. Como verificação de sanidade, também avaliamos métodos que utilizam incorporações extraídas apenas de palavras-chave. Embora as incorporações baseadas em palavras-chave gerem AUCs semelhantes, observamos uma queda acentuada no recall (0,53). Isso sugere que palavras-chave sozinhas não são suficientes para caracterizar relatórios de bugs EH, indicando a necessidade de análises textuais mais complexas. Conclusões: Até onde sabemos, este é o primeiro estudo a abordar o problema da rotulagem automática de bugs EH. Com base em nossos resultados,

podemos concluir que a combinação de técnicas de NLP e ML é promissora para automatizar a tarefa de rotulagem de bugs EH. Esperamos, em geral, que (i) nosso trabalho contribua para aumentar a conscientização sobre os bugs EH e (ii) que nosso conjunto de dados (disponível publicamente) sirva como um conjunto de dados de referência, abrindo caminho para trabalhos futuros. Além disso, nossas descobertas podem ser utilizadas para construir ferramentas que ajudem os mantenedores a identificar bugs EH durante o processo de triagem.

Palavras-chave: bug de tratamento de exceção; rotulagem automática de bugs; aprendizado de máquina; processamento de linguagem natural.

ABSTRACT

Context: Exception randling (EH) bugs stem from incorrect usage of exception handling mechanisms (EHM) and often incur severe consequences (e.g., system downtime, data loss, and security risk). Tracking EH bugs is particularly relevant for contemporary systems (e.g., cloud- and artificial intelligence based systems), in which the software's sophisticated logic is an additional threat to the correct use of the EHM. On top of that, bug reporters seldom can tag EH bugs — since it may require an encompassing knowledge of the software's EH strategy. Surprisingly, to the best of our knowledge, there is no automated procedure to identify EH bugs from report descriptions. Objective: First, we aim at evaluating the extent to which Natural Language Processing (NLP) and Machine Learning (ML) can be used to reliably label EH bugs using the text fields from bug reports (e.g., summary, description, and comments). Second, we aim at providing a reliably labeled dataset that the community can use in future endeavors. Overall, we expect our work to raise the community's awareness regarding the importance of EH bugs. Method: We manually analyzed 4,516 bug reports from the four main components of Apache's Hadoop project, out of which we labeled $\approx 20\%$ (943) as EH bugs. Then, we used word embedding techniques (Bag-of-Words and Term Frequency-Inverse Document Frequency (TF-IDF)) to summarize the textual fields of bug reports. Subsequently, we used these embeddings to fit four classes of ML methods and record their performance on unseen data. We have also evaluated whether considering only EH keywords is enough to achieve high predictive performance. Results: Our results show that the combination of NLP and ML techniques can label EH bugs reasonably well, achieving Receiver Operating Characteristics-Area Under The Curve (ROC-AUC) scores of up to 0.70 and recall ranging from 0.50 up to 0.62. As a sanity check, we also evaluate methods using embeddings extracted solely from keywords. While keyword-based embeddings yield similar AUC, we observe a steep decrease in recall (0.53). This suggests that keywords alone are not sufficient to characterize reports of EH bugs — and there is an avenue for more complex text analyses. Conclusions: To the best of our knowledge, this is the first study addressing the problem of automatic labeling of EH bugs. Based on our results, we can conclude that the combination of NLP and ML techniques sounds promising to automate the task of labeling EH bugs. Overall, we hope (i) that our work will contribute towards raising awareness around EH bugs; and (ii) that our (publicly available) dataset will serve as a benchmarking dataset, paving the way for follow-up works. Additionally, our findings can be used to build tools that help maintainers flesh out EH bugs during the triage process.

Keywords: exception handling bug; automatic bug labeling; machine learning; and natural language processing.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

API	Application Programming Interface
ASF	Apache Software Foundation
AUC	Area Under The Curve
AW	Available Words
BoW	Bag of Words
EH	Exception Handling
EHK	Exception Handling Keywords
EHM	Exception Handling Mechanism
HDFS	Hadoop Distributed Filesystem
IDE	Integrated Development Environment
KNN	K-nearest neighbors
LRC	Linear Regression Classification
LSI	Latent Semantic Indexing
ML	Machine Learning
MLP	Multilayer Perceptron
MNB	Multinomial Naive Bayes
NLP	Natural Language Processing
RFC	Random Forest Classification
ROC-AUC	Receiver Operating Characteristics-Area Under The Curve
SVC	Support Vector Classifier
TF-IDF	Term Frequency-Inverse Document Frequency
YARN	Yet Another Resource Negotiator

CONTENTS

1 INTRODUCTION

Exception handling (EH) is a forward error-recovery technique that allows us to anticipate abnormal situations. When a system reaches these abnormal states during runtime, it triggers a series of pre-defined recovery actions.

Besides improving robustness (SHAHROKNI; FELDT, 2013), EH enables the separation of error-handling code from regular code, enhancing software comprehensibility and maintainability (CHEN *et al.*, 2009; CACHO *et al.*, 2014a; CACHO *et al.*, 2014b). However, the way EH features are implemented in mainstream program languages (e.g., C#, Java, and Python) leads developers to create multiple control flows, making the software harder to debug (ROBILLARD; MURPHY, 2003; CHANG; CHOI, 2016) and posing new challenges to software testing (SINHA; HARROLD, 2000; ZHANG; ELBAUM, 2014; DALTON *et al.*, 2020; MARCILIO; FURIA, 2021; LIMA *et al.*, 2021).

Despite the importance of EH, several studies report that EH is often poorly understood, usually neglected, and insufficiently tested by developers (mostly by novice ones) (SHAH *et al.*, 2010; KECHAGIA; SPINELLIS, 2014; ZHANG; ELBAUM, 2014; ASADUZZAMAN *et al.*, 2016; GOFFI *et al.*, 2016; CHANG; CHOI, 2016; FILHO *et al.*, 2017). The combination of these factors creates a fertile ground for defects caused by the incorrect use of the EH mechanism (EHM), baptized "*exception handling bugs*" by Ebert *et al.* (2015). While EH was always a complex subject, (CHEN *et al.*, 2019a) recently argued that the vast space of potential error conditions and the sophisticated logic of modern systems (e.g., cloud-based, microservice-based, and big data-oriented) makes using EHMs correctly even harder, leaving modern software systems especially prone to EH bugs. In these complex systems, EH bugs may lead to dire consequences, such as system downtime, data loss, and security risk (ZHANG *et al.*, 2021). Given these potential risks, EH bugs must be quickly triaged (i.e., identified, prioritized, and assigned) and fixed.

The bug triage process is typically done by reading each bug report to better understand its nature (e.g., source, kind, and severity), prioritizing and assigning it to a maintainer who best fits (CATOLINO *et al.*, 2019). However, as the bug report backlog increases, the triage process becomes a time and resource-consuming task as well (PICUS; SERBAN, 2022; KÖKSAL; ÖZTÜRK, 2022). A straightforward solution to improve this process consists of enriching the bug report (before the triage starts) with informative labels to best characterize each reported bug. Nevertheless, this labeling task mostly relies on the bug reporter's knowledge, time,

and convenience which may lead to reliability information issues, calling for automatization.

Previous works on EH bugs have explored the relationship between EH and post-release defects by identifying, classifying, and quantifying the source of EH bugs (BARBOSA *et al.*, 2014; EBERT *et al.*, 2015; COELHO *et al.*, 2017; PáDUA; SHANG, 2017; EBERT *et al.*, 2020; SOUSA *et al.*, 2020) and investigating the existence of statistical relationships between them (MARINESCU, 2011; SAWADPONG *et al.*, 2012; MARINESCU, 2013; SAWADPONG; ALLEN, 2016; PáDUA; SHANG, 2018). These studies provide empirical evidence that discloses a substandard in EH implementation practices and how this phenomenon can impact several quality attributes (e.g., maintainability, reliability, and robustness) (MELO *et al.*, 2019). On a different note, a number of works focus on leveraging Machine Learning (ML) and Natural Language Processing (NLP) techniques to help in bug triage by performing automatic issue type classification (if bug or not) (PANDEY *et al.*, 2017; CHAWLA; SINGH, 2015; AUNG *et al.*, 2022), labeling the kind of bug (e.g., security and permission) (CHAWLA; SINGH, 2014; PETERS *et al.*, 2019; CATOLINO *et al.*, 2019; ELZANATY *et al.*, 2021), assigning bug severity (GOMES *et al.*, 2019; PICUS; SERBAN, 2022), estimating priority (TIAN *et al.*, 2015; UDDIN *et al.*, 2017), and suggesting the fixer (HU *et al.*, 2014; LEE *et al.*, 2017; CHEN *et al.*, 2019b; AUNG *et al.*, 2022). Surprisingly, however, there are no works on using ML and NLP to improve the triage of EH bugs.

In this study, we empirically evaluate the idea of automatically labeling EH bugs using ML classifiers and NLP techniques to extract features from bug report fields (e.g., summary, description, and comments). However, the use of such techniques to label EH bug reports poses challenges due to the lack of previously labeled datasets to build models and the existence of a class imbalance problem (i.e., EH bugs represent a small percentage of reported bugs).

To bridge this gap, we first built a manually labeled dataset from an existing dataset that contains 10 years of bug-fixing activity from the Apache Hadoop project. Thus, 4,516 bug reports were manually inspected and 943 (about 20%) of them were labeled as EH bugs. Next, we analyzed our dataset to determine whether the lack of attention given to EH, as reported in previous work, also occurs in bug-fixing activities. To this end, we compared EH and non-EH bugs with respect to their priorities, fixing time, number of comments in reports, and the number of changed test files in fix commits. Finally, we perform a controlled experiment combining four ML classifiers (Support Vector Classifier, Multinomial Naive Bayes, Linear Regression, and Random Forest) with two NLP strategies to extract features from the bug report

text (Bag of Words and TF-IDF). We also evaluate if using Bag of Words and TF-IDF only on keywords related to exception handling extracted from textual fields could improve the ML models' performance.

Our results show that the combination of NLP and ML techniques achieved good performance for the task of automatic labeling of EH bugs. The approach achieved scores of ROC-AUC metric up to 0.70. Additionally, considering only keywords related to EH, the ML models yield similar AUC, and we observe a steep decrease in recall (0.53). To the best of our knowledge, this is the first study addressing the task of automatic labeling of EH bugs.

2 BACKGROUND

2.1 What is an Exception?

The terms *failure*, *error*, *fault*, *defect*, and *bug* are frequently referred to in software testing literature. Although their meanings are related, there are important distinctions between these four concepts. The first three terms (failure, error, and fault) are well understood in the Dependable Computing and Fault Tolerance communities (AVIZIENIS *et al.*, 2004). A failure occurs when the system's external behavior does not conform to its specification. An error is a system's internal state, which in the absence of a proper system recovery action could lead it to failure. A fault is the adjudged or hypothesized cause of an error. A fault may remain dormant for a long period until activated by some event. A defect is a flaw in a software system that could lead it to behave erroneously or improperly, different from what is expected. Considering the source of software failure, the terms defect and fault can be seen as synonymous and interchangeable. The term bug is widely used by the developer's community to refer to a software defect, thus we adopt this term in this paper.

An exception is an event that models a state in which the normal flow of system execution cannot continue (KIENZLE, 2008). In order for the system to continue executing correctly, the flow of execution must deviate and an additional computation must be employed to deal with that situation (KNUDSEN, 1987). In reliable systems, an error can be modeled as an exception, as it rarely happens during system execution (GOODENOUGH, 1975; PARNAS; WüRGES, 1976). Exception handling provides a means to structure fault tolerance activities through error recovery (GARCIA *et al.*, 2001). However, exceptions can model other situations (MILLER; TRIPATHI, 1997), such as (i) deviation - the emergence of an invalid state, but which is allowed by the system; (ii) notification - information to the invoker of the operation that the state of the system has changed; and (iii) languages - other uses where the occurrence of the exception is rare rather than abnormal.

2.2 Java Exception Handling

In Java programming language, "an exception is an event, which occurs during the execution of a program, which disrupts the normal flow of the program's instructions" (GALLARDO *et al.*, 2014). When an error occurs inside a method, an exception is raised. In Java,

the raising of an exception is called *throwing*. Exceptions are represented as objects following a proper class hierarchy. Exceptions can be divided into two categories: checked and unchecked exceptions. Checked exceptions are all exceptions that inherit, directly or indirectly, from `Exception` class from `java.lang` package, except those that inherit, directly or indirectly, from `Error` or `RuntimeException` classes (both from `java.lang` package), named unchecked ones. Checked exceptions represent exceptional conditions that a robust application should anticipate and recover from. Unchecked exceptions represent an internal (`RuntimeException`) or an external (`Error`) exceptional conditions that the application usually cannot anticipate or recover from. In Java, the handling of checked exceptions is mandatory while the handling of unchecked exceptions is not.

When an exception is raised, the execution flow is interrupted and deviated to a specific point where the exceptional condition is handled. In Java, exceptions can be raised using the `throw` statement, signaled using the `throws` statement, and handled in the `try-catch-finally` blocks. The "`throw new E()`" statement is an example of *throwing* the exception `E`. The "`public void m() throws E`" is an example of how `throws` statement is used in the method declaration to indicate the signaling of exception `E`.

The `try` block is used to enclose the method calls that might throw an exception, also called protected region. If an exception occurs within the `try` block, that exception is handled by an exception handler associated with it. Handlers are associated with a `try` block by putting a `catch` block after it. A `try` block can be associated with multiples `catch` blocks. Each `catch` block catches a specific exception type and encloses the exception handler code. The `finally` block is optional, but whether declared always executes when the `try` block finishes, even if an exception occurs. Cleanup actions are usually coded within the `finally` block.

2.3 Exception Handling Bug

To better understand EH bugs, it is first necessary to precisely define when a bug is considered an EH bug or not. One of the most accepted definitions for EH bugs was given by Ebert *et al.* (2015): "*An Exception Handling Bug is a bug whose cause is related to exception handling. EH-bugs can occur when the exception is defined, thrown, propagated, handled, or documented; in the clean-up action of a protected region where the exception is thrown; when the exception should have been thrown or handled while it is not thrown or handled*". In this study, we choose the Ebert *et al.* (2015) definition of EH bug to support our manual labeling of

reported bugs as EH bug or not.

Identifying an EH bug is not an easy task. It requires inspecting the bug report fields (summary, description, and comments) to understand the source of a bug and if it complies with the EH bug definition. To illustrate this process, we present some examples of EH bugs from the Apache Hadoop project in the next paragraphs.

Sometimes, the information needed to classify the reported bug as an EH bug is easy to find in the bug report summary itself. It is exactly the case of the bug report `HDFS-13100`[1] from Hadoop's HDFS module (see Figure 1). In fact, the cause of `HDFS-13100` bug is the incorrect handling of two exceptions: `UnsurportedOperationExcetion` and `IllegalArgumentException`. The fix action addresses the bug by implementing the following rules: (i) if the required operation is not supported, the `UnsurportedOperationExcetion` must be thrown; and (ii) if the given parameter is not a legal one, the exception `IllegalArgumentException` must be thrown.

Hadoop HDFS / HDFS-13100

Handle IllegalArgumentException when GETSERVERDEFAULTS is not implemented in webhdfs.

Figure 1 – Summary content of `HDFS-13100` bug report.

In other cases, it is necessary to go beyond and also inspect the bug report description, as in the case of `MAPREDUCE-6156`[2] bug report of the Hadoop's MapReduce module (see Figure 2). The description of `MAPREDUCE-6156` bug gives us the idea that the handler (catch block) associated with the `IOException` is not dealing properly with the connection timeout variable. The fix provided by Hadoop's maintainers addresses exactly this problem.

There are cases in which inspecting only the report summary and description is not enough to classify the reported bug as an EH bug. In this case, it is necessary to go deeper and analyze the comments posted by the maintainers and the discussions between them. The `HDFS-1505`[3] bug report of Hadoop Distributed Filesystem (HDFS) module is an example of that (see Figure 3). It is possible to infer from the comments that the maintainers reached an understanding that the cause of the reported bug is the lack of throwing an exception to characterize the failure to save in all image directories.

[1] <https://issues.apache.org/jira/browse/HDFS-13100>
[2] <https://issues.apache.org/jira/browse/MAPREDUCE-6156>
[3] <https://issues.apache.org/jira/browse/HDFS-1505>

Description

The connect() function in the fetcher assumes that whenever an IOException is thrown, the amount of time passed equals "connectionTimeout" (see code snippet below). This is incorrect. For example, in case the NM is down, an ConnectException is thrown immediately - and the catch block assumes a minute has passed when it is not the case.

```
if (connectionTimeout < 0) {
    throw new IOException("Invalid timeout "
                          + "[timeout = " + connectionTimeout + " ms]");
  } else if (connectionTimeout > 0) {
    unit = Math.min(UNIT_CONNECT_TIMEOUT, connectionTimeout);
  }
  // set the connect timeout to the unit-connect-timeout
  connection.setConnectTimeout(unit);
  while (true) {
    try {
      connection.connect();
      break;
    } catch (IOException ioe) {
      // update the total remaining connect-timeout
      connectionTimeout -= unit;

      // throw an exception if we have waited for timeout amount of time
      // note that the updated value if timeout is used here
      if (connectionTimeout == 0) {
        throw ioe;
      }

      // reset the connect timeout for the last try
```

Figure 2 – Description of MAPREDUCE-6156 bug report.

Todd Lipcon added a comment - 17/Nov/10 20:39

here's a test that shows the problem.

If all of the image directories fail to saveNamespace, saveNamespace itself should probably throw an exception, no?

Eli Collins added a comment - 18/Nov/10 00:32

Patch looks good. Why comment out the three tests that use saveNamespaceWithInjectedFault?

Eli Collins added a comment - 18/Nov/10 00:33

I agree saveNamespace should throw an exception if it fails all to save to any image directory.

Figure 3 – Comments of HDFS-1505 bug report.

3 RELATED WORK

In this section, we describe the existing studies that are, somehow, related to our study. Although none of them focus on the problem to the automatic labeling of EH bugs, they comprise studies regarding EH bugs (Section 3.1) and automatic bug labeling (Section 3.2).

3.1 Studies on Exception Handling Bugs

The studies conducted by Barbosa *et al.* (2014) and Ebert *et al.* (2015) gather evidence that erroneous or improper usage of exception handling can lead to a series of fault patterns, named "exception handling bugs". This kind of fault refers to a bug in which the primary source is related to (i) the exception definition, throwing, propagation, handling, or documentation; (ii) the implementation of cleanup actions; and (iii) the wrong throwing or handling (i.e., when the exception should be thrown or handled and it is not). Barbosa *et al.* (2014) categorizes 10 causes of exception handling bugs, analyzing two open source projects, Apache Tomcat and Hadoop framework. Ebert *et al.* (2015) presents a comprehensive classification of exception handling bugs based on a survey of 154 developers and the analysis of 220 exception handling errors reported from two open-source projects, Apache Tomcat and Eclipse Integrated Development Environment (IDE).

Pádua and Shang (2017) conducted a study on the prevalence of exception-handling anti-patterns across 16 open-source projects (Java and C#). They claim that the misuse of exception handling can cause catastrophic software failures, including application crashes. They found that all 19 exception-handling anti-patterns taken into account in the study are broadly present in all subject projects but only 5 of them (unhandled exception, generic catch, unreachable handler, over-catch, and destructive wrapping) are prevalent.

Kechagia and Spinellis (2014) studied undocumented runtime exceptions thrown by the Android platform and third-party libraries. They mined 4,900 different stack traces from 1,800 apps looking for undocumented Application Programming Interface (API) methods with undocumented exceptions participating in the crashes. They found that 10% of crashes might have been avoided if the correspondent runtime exceptions had been properly documented.

Coelho *et al.* (2017) mined 6,000 stack traces from over 600 open-source projects issues on GitHub and Google Code searching for bug hazards regarding exception handling. Additionally, they surveyed 71 developers involved in at least one of the projects analyzed.

As a result of the mining phase, they found four bug hazards that may cause bugs in Android applications: (i) cross-type exception wrapping; (ii) undocumented unchecked exceptions raised by the Android platform and third-party libraries; (iii) undocumented check exceptions signaled by native C code; and (iv) programming mistakes made by developers. The survey results corroborate the stack trace findings, indicating that developers are unaware of frequently occurring undocumented exception handling behavior.

3.2 Studies on Automatic Bug Labeling

Chawla and Singh (2014) proposed an approach for automatic bug labeling by incorporating semantically similar terms present in the bug data. The work presents an automated technique for bug labeling using Term Frequency-Inverse Document Frequency (TF-IDF) and Latent Semantic Indexing (LSI). For the study, they selected bug reports from Google Chrome labeled with the following categories: security, regression, polish, and clean up, totalizing 4319 bug reports. The preprocessing included tokenization, stop-words removal, and stemming. Multinomial Naive Bayes was used for labeling. The Experimental study shows that there is an improvement in results with the addition of semantically similar words obtained from LSI in conjunction with the terms extracted using TF-IDF. The labeling accuracy is improved in two out of four categories with the addition of semantically similar terms.

To facilitate the screening of bugs, Catolino *et al.* (2019) analyzed 1280 bug reports of 119 popular projects. They proposed a novel taxonomy of bug types and an automated classification model to classify the reported bugs according to the defined taxonomy. They used Logistic Regression and analyzed the performance using F-measure, AUC-ROC, and Matthew's Correlation Coefficient (MMC). As a result, nine different types of bugs were highlighted and the proposed bug type classification model achieved an overall F-Measure, AUC-ROC, and MMC of 64%, 74%, and 72%, respectively, presenting a good performance for the bug type classification.

Elzanaty *et al.* (2021) presents an approach to automatically recover issue types in an industrial setting. In his work, a random sample of 951 issue reports from three repositories developed by Shopify were manually classified. The study trained four machine learning classifiers (K-nearest neighbors (KNN), Multinomial Naive Bayes (MNB), Support Vector Classifier (SVC), and Multilayer Perceptron (MLP)) to automatically label issue reports as defect-fixing or not using NLP-based features. As a result, the classifiers outperform random guessing (AUC values of 0.5271–0.8070) and Zero-R baselines (F1-score improvements of 0.31–

21.72 percentage points). When datasets from other projects are integrated to create a unique training sample, the models achieve performances equivalent to the intra-project classifiers. In the analysis, the SVC and MLP classification techniques improve the F1-score and AUC from within-design baselines in four out of six and two out of six experiments. The study highlights the combining NLP and ML techniques to classify missing issue types and lay the groundwork for adopting software analytics at Shopify.

Peters *et al.* (2019) proposed a way to reduce the mislabelling of security bug reports by developing a framework composed of a combination of Filtering And Ranking methods by text-based prediction models. The study evaluated 45.940 bug reports from Chromium and four Apache projects. The framework begins by finding security-related keywords from the security bug reports. Each security-related keyword is scored according to its frequency. After that, the authors removed nonsecurity reports with scores that are similar to the ones obtained by security bug reports. The remaining reports are used to build the prediction models. The analysis demonstrated that the proposed framework improves the performance of text-based prediction models for security bug reports in 90% of cases, mitigates the class imbalance issue, and reduces the number of mislabelled security bug reports by 38%.

4 THE EH-BUG DATASET

Our EH-Bug dataset was derived from an existing Bug-Fixing dataset. In this section, we first describe the original dataset (Section 4.1) and then we describe the EH-Bug dataset itself (Section 4.2).

4.1 The Original Dataset

Vieira *et al.* (2019) propose a dataset comprising a set of 10-years bug-tracking information from 55 open-source projects from the Apache ecosystem. We describe in this section the Vieira *et al.* (2019) data collection methodology and the description of the dataset itself.

The Vieira *et al.* (2019) dataset was created using data extracted from the official Jira[1] and Git[2] repositories of the Apache Software Foundation (ASF). First, the Jira repository was mined selecting issues labeled as "Bug" with `CLOSED` or `RESOLVED` status and with the "Fixed" resolution status. The mining process targeted bug reports created and fixed between 2009-01-01 and 2019-01-02. They used Python Jira[3] library to automate the mining process. Second, they used the bug report ID of mined issues from Jira to mine Git repository using Pydriller[4] (SPADINI *et al.*, 2018) framework to retrieve the respective fixing commits, resulting in the first dataset they called `snapshot`.

Using the list of retrieved issues IDs from Jira, they mined other datasets. The first one was the `change-log` dataset, which contains all the changes made in each bug report during the considered time period. The second set was the `comment-log` dataset, which contains all the comments on each bug report posted during the same period of time. The last one was called of `commit-log`, which contains a dataset with detailed information about fixing commits.

Finally, Vieira *et al.* (2019) performed a pre-processing in the text fields (i.e, summary, description, comments, and commit messages) of each bug report using the NLTK[5], a Python library for Natural Language Processing, to extract and store the 1,000 most frequent words and their respective frequencies in the dataset.

Overall, Vieira *et al.* (2019) dataset provides information under two perspectives

[1] <https://issues.apache.org/jira>
[2] http://gitbox.apache.org
[3] https://jira.readthedocs.io/
[4] <https://github.com/ishepard/pydriller>
[5] <https://www.nltk.org/>

(static and dynamic) we explain in the following.

Static Perspective. For each bug report, 53 attributes are available, divided into data points collected from Jira and from Git. Additionally, the attributes were also classified according to the nature of the information they represent: general (standard information), text (textual information), time (time-related information), versioning (system version-related information), summation (fields that store counting information), link (bug dependencies), and source (source code related information). The complete list of static perspective (`snapshot`) dataset fields can be found in Table 1.

Dynamic Perspective. The bug reports contain attributes with immutable information such as the `CreationDate` and `Key` (identifier). Other attributes, such as `AffectsVersions` and `Assignee`, may not be required and may change during the lifetime of the report. The Bug Report is constantly changing and updating until it is resolved. The dynamic dataset perspective represents those times when the report changes, when new information is added to the report, or a field changes, such as status or priority change; a new comment is added; a new employee starts to be responsible for fixing the problem. The dynamic dataset is composed of three files: (i) `changelog`: This dataset stores every modification that ever happened on every Jira report field. The data fields are shown in Table 2 and they were mined from Jira; (ii) `comment-log`: This dataset stores information about each comment related to its report. These data fields, mined from Jira, are shown in Table 3; and (iii) `commit-log`: A number of bug reports is related to some commits that fixes that bug. This dataset stores commit information related to each report that has one commit. The dataset entries bring detailed information about each file modified by bug-fix commits. The data fields are shown in Table 4.

4.2 Our Dataset

Our EH-Bug dataset was derived from Vieira *et al.* (2019) dataset (see Section 4.1) considering only the Apache Hadoop project. Hadoop is an open-source framework developed by

Table 1 – The `snapshot` dataset fields, with 53 attributes acquired from Jira and Git.

From	Type	Field
Jira (30)	General (10)	`Project`
		`Owner`
		`Manager`
		`Category`
		`Key`
		`Priority`
		`Status`
		`Reporter`
		`Assignee`
		`Components`
	Link (2)	`InwardIssueLinks`
		`OutwardIssueLinks`
	Summation (4)	`NoComments`
		`NoWatchers`
		`NoAttachments`
		`NoAttachedPatches`
	Text (3)	`SummaryTopWords`
		`DescriptionTopWords`
		`CommentsTopWords`
	Time (8)	`CreationDate`
		`ResolutionDate`
		`FirstCommentDate`
		`LastCommentDate`
		`FirstAttachmentDate`
		`LastAttachmentDate`
		`FirstAttachedPatchDate`
		`LastAttachedPatchDate`
	Versioning (2)	`AffectsVersions`
		`FixVersions`
Git (24)	Text (1)	`CommitsMessagesTopWords`
	Versioning (1)	`HasMergeCommit`
	Summation (3)	`NoCommits`
		`NoAuthors`
		`NoCommitters`
	Time (4)	`AuthorsFirstCommitDate`
		`AuthorsLastCommitDate`
		`CommittersFirstCommitDate`
		`CommittersLastCommitDate`
		`NonSrcAddFiles`
		`NonSrcDelFiles`
		`NonSrcModFiles`
		`NonSrcAddLines`

Table 2 – The `changelog` dataset fields.

Field	From	Type
Jira (9)	General (6)	`Project`
		`Manager`
		`Category`
		`Key`
		`Author`
		`Field`
	Time (1)	`ChangeDate`
	Text (2)	`From`
		`To`

Table 3 – The `comment-log` dataset fields.

Field	From	Type
Jira (7)	General (5)	`Project`
		`Manager`
		`Category`
		`Key`
		`Author`
	Time (1)	`CommentDate`
	Text (1)	`Content`

the Apache Software Foundation for distributed and scalable computing. This distributed system allows the storage and processing of large datasets across clusters of computers and is designed to detect and handle faults, providing a highly reliable service (WHITE, 2015). The Hadoop architecture comprises four main components: (i) Core: which provides the utility package to support other Hadoop modules; (ii) MapReduce: a programming model for storage and data processing. Its parallel programming comes into its own in large-scale data analysis; (iii) Hadoop Distributed Filesystem (HDFS): a distributed filesystem that runs on clusters designed for storing very large files and providing high-throughput access; (iv) Yet Another Resource Negotiator (YARN): a framework for job scheduling/monitoring and cluster resource management. Provides APIs for requesting and working with cluster resources hiding the resource management details from the user. YARN was introduced to improve the MapReduce implementation, but its functions allowed other distributed computing projects and paradigms to be aggregated as well. Table 5 shows the name, year of the first release, and number of bugs for each component considering both filtering steps we explain later.

Table 4 – The `commit-log` dataset fields.

Field	From	Type
Jira (4)	General (4)	`Project`
		`Manager`
		`Category`
		`Key`
Git (18)	Versioning (2)	`CommitHash`
		`IsMergeCommit`
	General (2)	`Author`
		`Committer`
	Time (2)	`AuthorDate`
		`CommitterDate`
	Text (1)	`CommitMessageTopWords`
	Source (11)	`FileName`
		`FilePath`
		`ChangeType`
		`IsSrcFile`
		`IsTestFile`
		`AddLines`
		`DelLines`
		`NoMethods`
		`LoC`
		`CyC`
		`NoTokens`

Table 5 – Target components of Hadoop project.

Category	Hadoop Component	1st Release	#Bugs 1st	#Bugs 2nd
big-data (4)	Core	2006	2861	1105
	YARN	2012	2090	1017
	HDFS	2009	3214	1504
	MapReduce	2009	2210	890

Hadoop (and consequently its components) has been chosen because it has a set of well-documented bug reports. Furthermore, Hadoop is widely used and incorporated in a large number of companies and their products. Its commercial support is available on large scale from companies such as EMC, IBM, Microsoft, and Oracle (WHITE, 2015).

The methodology we used to create the EH-Bug dataset is depicted in Figure 4. In the **1st filtering**, we select from the original `snapshot` dataset only the records related to the four components of Hadoop, resulting in a total of **10,375** bug reports. After that, we try to get more probably EH bugs by applying a **2nd filtering** over the set of the selected bug reports. In this filtering, we select only reported bugs that in at least one text field (summary, description, comments, and commits message) have any EH-related keyword. We build our set of EH keywords based on the (EBERT *et al.*, 2015) study, which considers relevant radicals for EH-related keywords, such as "catches", "thrown", and "raises" and believes that these keywords are likely linked to EH issues. Thus, our final set of EH-related keywords is ["catch", "caught", "handl", "exception", "throw", "rais", "signal"]. This 2nd filtering results in **4,516** bug reports.

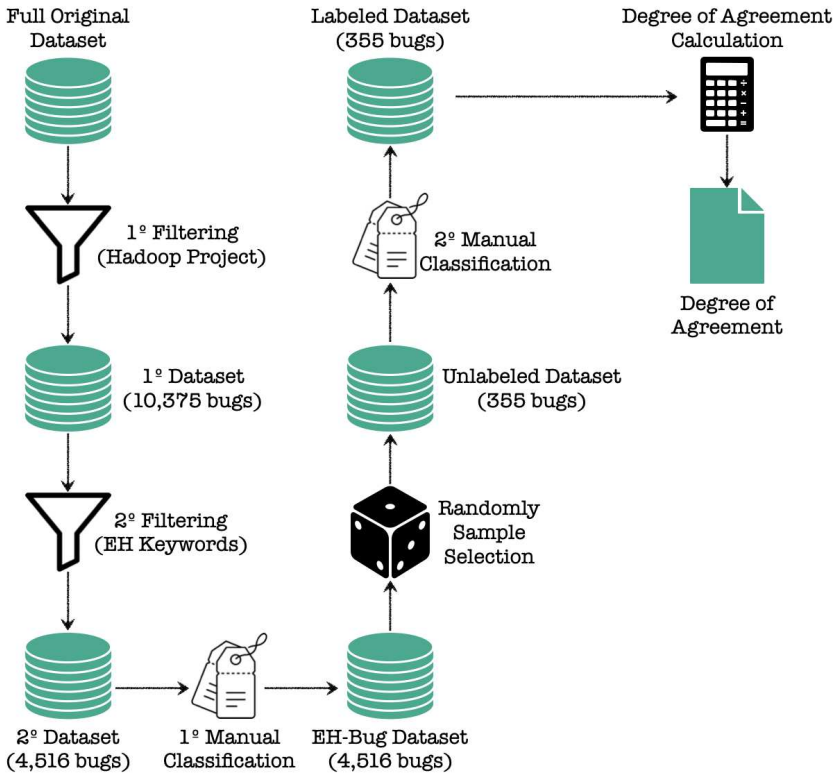

Figure 4 – Dataset creation methodology flow covering all the steps to the final EH bug dataset.

All bug reports resulting from the second filtering were manually inspected and

classified into two categories: EH bug and non-EH bug. The attribute "Type" was created in the dataset and assigned 1.0 for the EH bug and 0.0 for the non-EH bug. This manual labeling was performed by the first author of this study taking into account information available in all text fields (summary, description, comments, and commits message) of each report. As a result, 943 ($\approx 20\%$) were labeled as EH bugs and 3,573 were labeled as non-EH bugs.

After the first manual classification, we performed an evaluation to assess the classification reliability. To do that, we performed a second manual classification and calculate the level of agreement between them. The second manual classification was performed by another independent author of this study on a random-selected significant sample from the bug reports under consideration. The sample size was computed considering the following statistical constraints: confidence level of 95% and margin of error of 5%. Considering the population of 4,573 reports and the statistical constraints, the significant sample size was computed as 355 bug reports.

We used Cohen's Kappa coefficient (COHEN, 1960) to measure the level of agreement between the two labelers. The Kappa coefficient can be computed using the formula: $(Pc - Pe)/(1 - Pe)$. Where Pc is the proportion of units for which the labelers agreed and Pe is the proportion of units for which agreement is expected by chance. Table 6 provides an interpretation of Cohen's Kappa coefficient.

Table 6 – Cohen's kappa score interpretation.

Kappa Statistic	**Strength of Agreement**
<0.00	Poor
[0.00, 0.20]	Slight
[0.21, 0.40]	Fair
[0.41, 0.60]	Moderate
[0.61, 0.80]	Substantial
[0.81, 1.00]	Almost Perfect

In our evaluation, we obtained a Cohen's kappa score of 0.673. Thus, according to Table 6, Cohen's kappa score obtained (0.673) is classified as a substantial level of agreement, being a result considered relevant to guarantee the reliability of our dataset.

4.3 Our Dataset Analysis

In this section, we will examine our dataset to assess whether developers approach EH bug-fixing in the same manner as they approach other bug-fixing tasks regarding priority, fixing time, discussion, and testing. We first establish the analysis design, with the overall goal, research questions, and methodology, followed by the results and answers to posed research questions.

4.3.1 Goal and Research Questions

As we mentioned early, previous studies have suggested that developers often pay less attention to exception handling (EH) design and code compared to other design and code parts (SHAH *et al.*, 2010; KECHAGIA; SPINELLIS, 2014; ZHANG; ELBAUM, 2014; ASADUZZAMAN *et al.*, 2016; GOFFI *et al.*, 2016; CHANG; CHOI, 2016; FILHO *et al.*, 2017). In this analysis, we aim to investigate whether this phenomenon also occurs during bug-fixing activity in the Hadoop project. By controlling the EH bug fields of our dataset (priority, bug-fixing time, number of comments, and number of test files changed) with its non-EH bug fields counterpart, we can reason how EH bug fixing can differ from the activity to fix other types of bugs. Hence, we asked the following research questions:

RQ1. *To what extent are EH bugs prioritized compared to non-EH bugs*?

To gain a better understanding of whether EH bugs receive less attention than other types of bugs, we will compare the extent to which EH bugs are assigned a lower/higher priority and require more/less time to be resolved compared to non-EH bugs.

RQ2. *To what extent are EH bugs discussed compared to non-EH bugs*?

We consider the number of comments posted in bug reports as a proxy for developers' discussions in bug-fixing tasks. Based on that, we compare the extent to which EH bugs have more/less discussion compared to non-EH bugs.

RQ3. *To what extent are EH bugs tested compared to non-EH bugs*?

Finally, we use the number of test files modified in the bug-fixing commits as an indicator of the developers' level of commitment to testing the fixed code. Then, we use this information to compare the extent to which EH bugs fixed are more/less tested compared to non-EH fixed ones.

To answer the research questions, first, we split the dataset into two groups: EH bugs and non-EH bugs. For each specific RQ, we apply hypothesis tests to look at the difference between both groups, considering specific dataset fields. To answer RQ1, we evaluate the priority and the fixing time (*i.e.*, the time between the creation and resolution of the report); RQ2, the number of comments; and RQ3, the number of changed test files (i.e., the sum of deleted, added and modified test files). We use the Mann-Whitney U test, value of $\alpha = 0.05$, and also compute the Cohen's delta effect size for each result. We have formalized both null and alternative hypotheses for each RQ.

For this analysis, we perform a few data processing steps. First, we transform the priority fields (originally reported as words, as seen in Table 8) to ordinal variables, from 1 (lower priority) to 5 (highest priority). We also remove some reports based on two rules: i) reports resolved in less than 15 minutes after their creation (4 reports); and ii) reports with no associated commit (817 reports). These filter rules are based on another work (VIEIRA *et al.*, 2022) that uses the same dataset discussed in Section 4.1. Based on a sample of 300 bug reports, the authors verify that 80% of the filtered-out bugs fall in one of the cases: duplicated, already resolved by another report, created with a solution (report to document the bug only, with no discussion purposes or bug resolution details), discovered later that was not a bug or reports asking for documentation updates. Finally, we removed some outliers that were three standard deviations away from the mean.

4.3.2 Results

Table 7 shows the hypothesis tests results, p-value, and effect size values for each hypothesis established in the RQs.

4.3.2.1 RQ1. To what extent are EH bugs prioritized compared to non-EH bugs?

Summary of RQ1: The EH bugs are significantly (i) less prioritized and (ii) take more time to be fixed than non-EH bugs.

To answer this question, we have formulated two groups of hypotheses, considering the priority level and bug-fixing time of EH and non-EH bugs.

Table 7 – Summary of hypothesis statement, the statistics test, and the Cliff's Delta effect size results. The symbols ✓ and ✗ indicate the result of the null hypothesis test (✓ fail to reject, and ✗ reject). The Cliff's Delta effect size interpretation: negligible = $[0, 0.147)$, small = $[0.147, 0.33)$, medium = $[0.33, 0.474)$, and large = $[0.474, 1]$.

MW Hypothesis	p-value	Effect Size
$\mathscr{H}_0^A$: EH_PRIORITY = NON_EH_PRIORITY (✗) $\mathscr{H}_1^A$: EH_PRIORITY > NON_EH_PRIORITY (✗) $\mathscr{H}_2^A$: EH_PRIORITY < NON_EH_PRIORITY (✓)	5.923×10^{-4}	0.14993 (small)
$\mathscr{H}_0^B$: EH_FIXING-TIME = NON_EH_FIXING-TIME (✗) $\mathscr{H}_1^B$: EH_FIXING-TIME > NON_EH_FIXING-TIME (✓) $\mathscr{H}_2^B$: EH_FIXING-TIME < NON_EH_FIXING-TIME (✗)	6.066×10^{-3}	0.1238 (negligible)
$\mathscr{H}_0^C$: EH_COMMENTS = NON_EH_COMMENTS (✓) $\mathscr{H}_1^C$: EH_COMMENTS > NON_EH_COMMENTS (✗) $\mathscr{H}_2^C$: EH_COMMENTS < NON_EH_COMMENTS (✗)	7.468×10^{-1}	0.0030 (negligible)
$\mathscr{H}_0^D$: EH_TEST = NON_EH_TEST (✓) $\mathscr{H}_1^D$: EH_TEST > NON_EH_TEST (✗) $\mathscr{H}_2^D$: EH_TEST < NON_EH_TEST (✗)	7.704×10^{-1}	0.0133 (negligible)

The first group of hypotheses contains the null hypothesis ($\mathscr{H}_0^{\mathscr{A}}$), stating that there is no difference in priority level between EH bugs and non-EH bugs. The alternative hypotheses, on the other hand, assume that EH bugs have either a higher ($\mathscr{H}_1^{\mathscr{A}}$) or a lower ($\mathscr{H}_2^{\mathscr{A}}$) level of priority than non-EH bugs.

The second group of hypotheses contains the null hypothesis ($\mathscr{H}_0^{\mathscr{B}}$) stating that there is no difference in bug-fixing time between EH bugs and non-EH bugs. The alternative hypotheses, in this case, assume that EH bugs have either a higher ($\mathscr{H}_1^{\mathscr{B}}$) or a lower ($\mathscr{H}_2^{\mathscr{B}}$) bug-fixing time than non-EH bugs. Table 7 (two first rows) shows the statistical test results for both groups of hypotheses.

In our dataset, a bug report can receive five different levels of priority (see Table 8): `Trivial`, `Minor`, `Major`, `Critical`, and `Blocker`. Table 9 shows the priority distribution of both groups of bugs (EH and non-EH bugs) and also the total and the percentile of each priority group. Looking at Table 9, one can see that EH bugs have less percentage of higher priority bugs

Table 8 – Bug report priority classification in Jira plataform.

Priority	Description
`Blocker`	Highest priority. Indicates that this issue takes precedence over all others.
`Critical`	Indicates that this issue is causing a problem and requires urgent attention.
`Major`	Indicates that this issue has a significant impact.
`Minor`	Indicates that this issue has a relatively minor impact.
`Trivial`	Lowest priority.

(7.41%) when compared with non-EH bugs (12.12%). Additionally, it is possible to see that EH bugs tend to have a greater percentage of lower-priority bugs when compared with non-EH bugs. This perception is confirmed by the statistical test results that reject the null hypothesis $\mathscr{H}_0^{\mathscr{A}}$, accepting the alternative hypothesis $\mathscr{H}_2^{\mathscr{A}}$. This indicates that not only the priority of EH and non-EH bugs are statistically different but also that the non-EH bugs are statistically more prioritized than EH bugs. In fact, the effect size shows that the average priority of EH bugs is 0.14993 standard deviations lower than the average priority of non-EH bugs.

Table 9 – Distribution of EH and non-EH bugs priority.

Priority	EH Bugs		Non-EH Bugs	
	Number	(%)	Number	(%)
`Blocker`	59	07.41	352	12.12
`Critical`	117	14.69	416	14.33
`Major`	482	60.55	1732	59.66
`Minor`	119	14.94	348	11.98
`Trivial`	19	02.38	55	01.89

Table 10 presents the bug-fixing time descriptive statistics for EH and non-EH bugs, while Fig 5 shows the boxplot of bug-fixing time. When comparing the boxplots, it is possible to see that the interquartile EH bug-fixing time is larger than the non-EH bug-fixing time. Additionally, all statistics of EH bugs in Table 10 are greater than non-EH bugs. This perception is confirmed by the statistical test results that reject the null hypothesis $\mathscr{H}_0^B$ and accept $\mathscr{H}_1^B$. This indicates that the bug-fixing times of EH and non-EH bugs are statistically different and

non-EH bugs are statistically fixed faster than EH bugs. The effect size shows that the average bug-fixing time of EH bugs is 0.1221 standard deviations lower than the average bug-fixing time of non-EH bugs.

Table 10 – Descriptive statistics results for EH and non-EH bugs concerning the lag time in bug fixing activities.

Bug Category	**Fixing Time (days)**				
	Mean	**Minimum**	**Maximum**	**Median**	**Std. Deviation**
EH	58.80	0.03	615.77	14.31	111.16
Non-EH	46.60	0.00	669.43	9.82	94.92

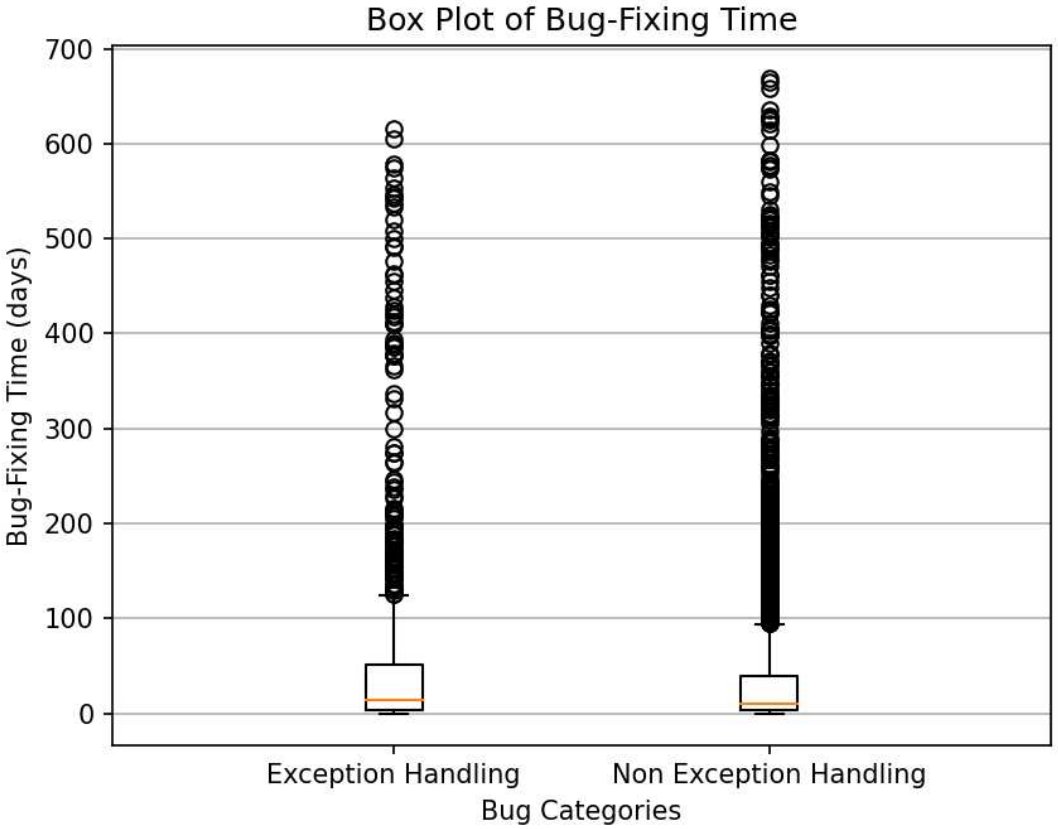

Figure 5 – Bug-fixing time boxplot. When comparing the boxplots, it is possible to see that the interquartile EH bug-fixing time is larger than the non-EH bug-fixing time.

4.3.2.2 *RQ2. To what extent are EH bugs discussed compared to non-EH bugs?*

Summary of RQ2: The EH bugs are not significantly less discussed than non-EH bugs.

To answer this question, we have formulated one group of hypotheses, considering the number of comments of EH and non-EH bugs.

The set of hypotheses includes the null hypothesis ($\mathscr{H}_0^{\mathscr{C}}$), which suggests that there is no difference in the number of comments between EH and non-EH bug reports. Conversely, the alternative hypotheses propose that EH bugs have either a higher ($\mathscr{H}_1^{\mathscr{C}}$) or a lower ($\mathscr{H}_2^{\mathscr{C}}$) number of comments compared to non-EH bugs.

Table 11 – Descriptive statistics results for EH and non-EH bugs concerning the number of comments.

Bug Category	**Number of Comments**				
	Mean	**Minimum**	**Maximum**	**Median**	**Std. Deviation**
EH	20.15	4	64	17	11.36
Non-EH	20.19	2	64	17	11.26

Table 11 presents the descriptive statistics for the number of comments posted in EH and non-EH bug reports, while Figure 6 shows the boxplot of the number of comments. Upon comparing the boxplots and statistics, it is evident that they are very similar. This observation is confirmed by the statistical test results that failed to reject the null hypothesis $\mathscr{H}_0^{C}$. Therefore, we can assume that the number of comments in EH bug reports is not statistically different from non-EH bug reports. Additionally, the computed effect size between the two groups, EH and non-EH bugs, is very low, 0.0030, almost negligible.

4.3.2.3 *RQ3. To what extent are EH bugs tested compared to non-EH bugs?*

Summary of RQ3: The EH bugs are not significantly less tested than non-EH bugs.

To answer this question, we have formulated one group of hypotheses, considering the number of changed test files of EH and non-EH bugs.

The group of hypotheses contains the null hypothesis ($\mathscr{H}_0^{\mathscr{D}}$) stating that there is no difference in the number of changed test files between EH and non-EH bugs. The alternative

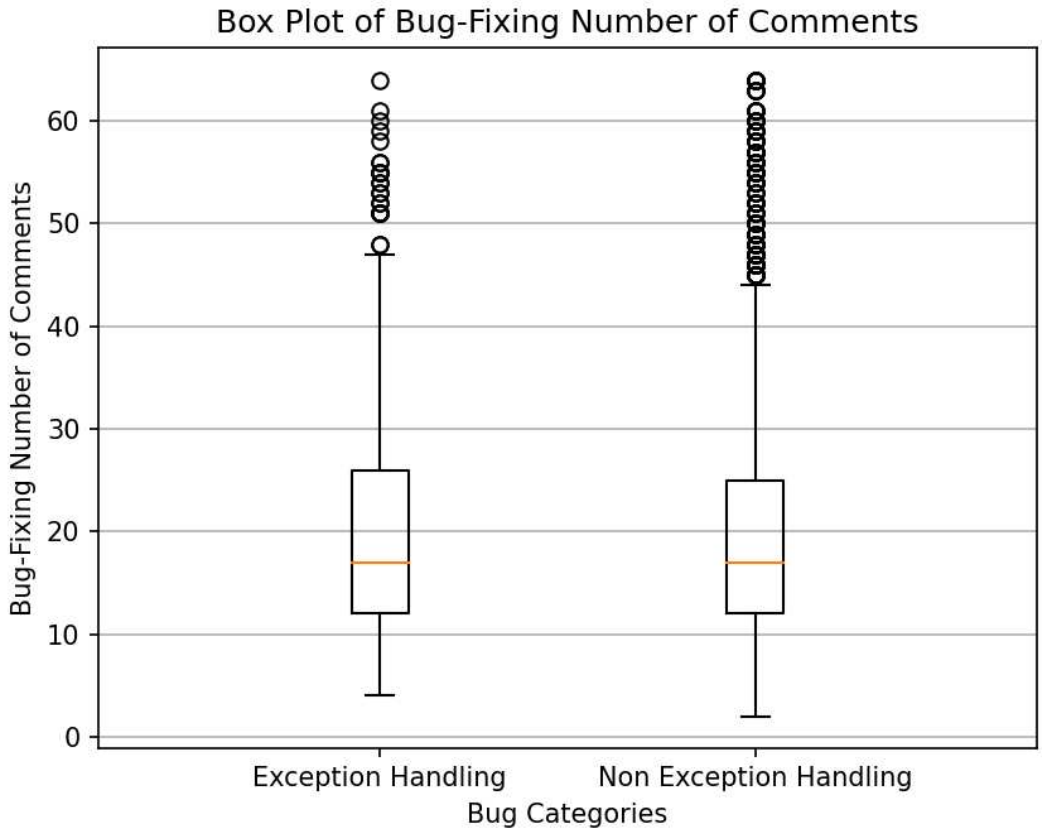

Figure 6 – Number of comments boxplot. Upon comparing the boxplot, it is evident that they are very similar.

hypotheses, on the other hand, assume that EH bugs have either a higher ($\mathscr{H}_1^{\mathscr{D}}$) or a lower ($\mathscr{H}_2^{\mathscr{D}}$) number of changed test files than non-EH bugs.

Table 12 – Descriptive statistics results for EH and non-EH bugs concerning the number of test files changed.

Bug Category	**Number of Changes**				
	Mean	**Minimum**	**Maximum**	**Median**	**Std. Deviation**
EH	1.138	0	12	1	1.51
Non-EH	1.119	0	12	1	1.45

Table 12 presents descriptive statistics for the number of test files changed in fixing commits of both EH and non-EH bugs. Meanwhile, Figure 7 displays a boxplot of the number of test files changed. Upon comparing the boxplots and statistics, it is evident that they are

very similar. This observation is confirmed by the statistical test results that failed to reject the null hypothesis $\mathscr{H}_0^D$. Therefore, we can assume that the number of test files changed in fixing commits of EH bugs is not significantly different from non-EH bugs. The computed effect size between the two groups, EH and non-EH bugs, is almost negligible at 0.0133.

Figure 7 – Boxplot of the number of test files changed in fixing commits. Upon comparing the boxplots, it is evident that they are very similar.

5 THE EH-BUG CLASSIFICATION MODEL

In this chapter, we describe the method to automatically label EH bugs using Machine Learning and Natural Language Processing techniques, along with the obtained results.

5.1 Method Description and Goal

The manual labeling of EH bugs is time-consuming, demanding the reading, understanding, and discussion of the bug report. Despise not being a trivial task, all conclusion around the EH bug classification is based on the report's textual fields: summary, description, and comments. These fields are usually presented in all bug reports, even though their "quality" may vary (*i.e.*, how they are detailed or faithful to the actual bug), impacting the task's challenge.

Once we have the labeled dataset and the necessary fields to classify an EH bug, we have a good setup to automate the EH bug classification task using machine learning. This section describes exploring machine learning models to identify bug reports as EH ones. We verify the feasibility of this model by testing different ML algorithms, NLP techniques, and how complex the task is, evaluating how much textual detail is necessary to achieve satisfactory results.

5.2 Experiment Design

All models use bug reports' textual fields as machine learning input: the content of summary, description, and comments. We test different machine learning and NLP techniques to evaluate the automatic EH classification. We use four models - Support Vector Classifier (SVC), Multinomial Naive Bayes (MNB), Linear Regression Classification (LRC), and Random Forest Classification (RFC) - and two different NLP encoding - Bag of Words (BoW, where the document corpus is converted in an array containing each text token/word count) and Term Frequency-Inverse Document Frequency (TF-IDF, which weights the relevance of each token/word in the document) (JONES, 1988). We also evaluate two different sets of words: i) All available Words (AW, containing all text from the report's textual fields) and ii) Exception Handling Keywords (EHK, where the keywords are `catch`, `caught`, `handl`, `exception`, `throw`, `rais` and `signal`), as defined by (EBERT *et al.*, 2015). The idea is to verify how complex the EH Bugs classification problem is and if it is feasible to identify them only using these specific keywords rather than using all available words (AW). Combining all these options, we have 16

Table 13 – Hyper-parameters Grid-search

Model	Hyper-parameters Grid-Search
Multinomial Naive Bayes	α: {0.00001, 0.00005, 0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05, 1.0, 5. 10,15, 20, 25, 30, 35, 40};
Logistic Regression	C: {0.001,0.005,0.01,0.05, 0.1, 0.5, 1, 5, 10 }; penalty: {'l1', 'l2'}; solver: {'liblinear', 'lbfgs'};
Support Vector Classifier	C: [0.01, 0.05, 0.1, 0.5, 1, 5, 10]
Random Forest	n_estimators: [5, 30, 50, 75, 100, 150, 200] max_depth: [4, 5, 6, 7, 8, None]

results based on the combination of four models, two different NLP encoding, and two sets of features ($4 \times 2 \times 2 = 16$).

We also train the models using 10-fold cross-validation and perform a grid search for the models' hyper-parameters. Table 13 presents the space search of each model. We use the nomenclature of the scikit-learn package for the parameters and suggest the official documentation[1] for more details about their meaning.

5.2.1 Results

Table 14 presents the average and standard deviation values for several metrics of the 10-fold runs. We highlight the best results of each metric in **boldface**.

Considering all the results, we highlight the results obtained by using BoW+LRC+AW. This combination provides the best F1 and ROC-AUC results, with values of accuracy and recall higher compared to the majority of results. We argue that recall is the major concern due to the main interest being to identify the majority of EH bugs. The combination BoW+MNB+AW provides the highest recall value, but presents one of the lowest precision values. Hence, we highlight the combination of BoW+LRC+AW for presenting the best balance between recall and precision compared to the other results.

[1] <https://scikit-learn.org/stable/modules/classes.html>

NLP (Tokens)	Models	Precision	Recall	Accuracy	ROC AUC	F1
BoW (AW)	SVC	0.49 ± 0.1	0.56 ± 0.1	0.78 ± 0.0	0.70 ± 0.0	0.52 ± 0.0
	MNB	0.28 ± 0.1	$\mathbf{0.62 \pm 0.2}$	0.57 ± 0.1	0.59 ± 0.1	0.37 ± 0.1
	LRC	0.52 ± 0.1	0.55 ± 0.1	0.79 ± 0.0	$\mathbf{0.70 \pm 0.0}$	$\mathbf{0.53 \pm 0.0}$
	RFC	0.43 ± 0.1	0.20 ± 0.1	0.78 ± 0.0	0.56 ± 0.0	0.27 ± 0.1
TF-IDF (AW)	SVC	0.60 ± 0.1	0.39 ± 0.2	0.81 ± 0.0	0.66 ± 0.1	0.45 ± 0.1
	MNB	0.51 ± 0.4	0.04 ± 0.0	0.80 ± 0.0	0.52 ± 0.0	0.08 ± 0.1
	LRC	0.65 ± 0.1	0.40 ± 0.2	0.83 ± 0.0	0.67 ± 0.1	0.47 ± 0.1
	RFC	0.40 ± 0.1	0.18 ± 0.0	0.77 ± 0.0	0.55 ± 0.0	0.25 ± 0.0
BoW (EHK)	SVC	$\mathbf{0.72 \pm 0.1}$	0.29 ± 0.1	0.83 ± 0.0	0.63 ± 0.0	0.41 ± 0.0
	MNB	0.53 ± 0.1	0.53 ± 0.1	0.80 ± 0.0	0.70 ± 0.1	0.52 ± 0.1
	LRC	0.70 ± 0.1	0.33 ± 0.1	$\mathbf{0.83 \pm 0.0}$	0.65 ± 0.0	0.45 ± 0.0
	RFC	0.62 ± 0.0	0.44 ± 0.1	0.83 ± 0.0	0.68 ± 0.0	0.51 ± 0.0
TF-IDF (EHK)	SVC	0.68 ± 0.1	0.23 ± 0.1	0.82 ± 0.0	0.60 ± 0.1	0.33 ± 0.2
	MNB	0.00 ± 0.0	0.00 ± 0.0	0.79 ± 0.0	0.50 ± 0.0	0.00 ± 0.0
	LRC	0.68 ± 0.1	0.25 ± 0.1	0.82 ± 0.0	0.61 ± 0.1	0.35 ± 0.2
	RFC	0.58 ± 0.0	0.41 ± 0.1	0.81 ± 0.0	0.67 ± 0.0	0.48 ± 0.1

Table 14 – The EH models for classification results.

We also evaluate the extent to which fine-grained text embeddings help classify EH bugs. Put simply: is the rich textual content of bug reports useful, or is focusing on a few keywords enough? To answer this question, we evaluate the effect of using AW over EHK as model inputs. More specifically, given a combination of a machine learning algorithm (RFC, SVC, MNB, and LRC) and an NLP word embedding (BoW or TF-IDF), we compare the recall performance in each test fold obtained using EHK and AW — subtracting the first from the latter. We repeat the experiment 10 times (using 10-fold cross-validation) to compute the average treatment effect of using AW over EHK. If using AW is consistently better than using EHK, we should see a positive effect. Figure 8 shows that using AW over EHK usually results in a positive effect. The only exception occurs with random forests. A plausible explanation is the higher dimension of AW embeddings can be harmful when training individual classification trees (XU *et al.*, 2012).

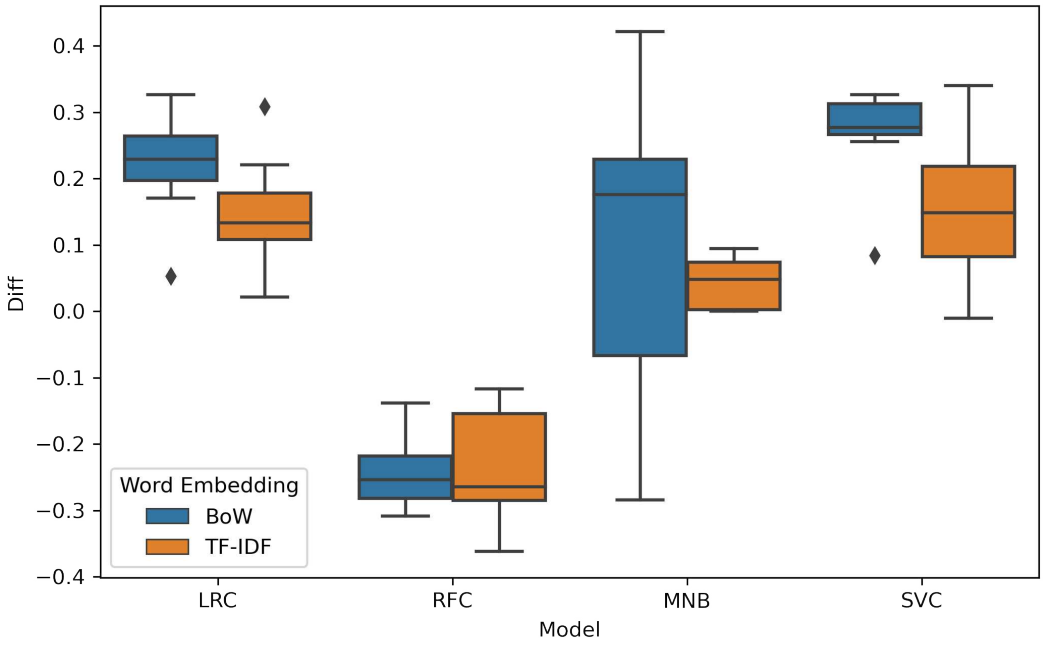

Figure 8 – Effect measure on recall of using AW × EHK on four models.

6 DISCUSSION, IMPLICATIONS, AND THREATS TO VALIDITY

6.1 Discussion and Implications

In this section, we discuss our results (Section 6.1.1), their implications for both, researchers (Section 6.1.2) and practitioners (Section 6.1.3), and threats to validity (Section 6.2).

6.1.1 Overall Discussion

The dataset allowed us to analyze how the eh-bugs are fixed compared to non-eh bugs. As mentioned earlier, many studies report that EH is often poorly understood, usually neglected, and insufficiently tested by developers (ASADUZZAMAN *et al.*, 2016; GOFFI *et al.*, 2016; CHANG; CHOI, 2016; FILHO *et al.*, 2017). With the dataset, we may verify if this is also reflected in the bug-fixing process. Our results indicate that even if the eh-bugs reports are less prioritized and demand more time to be fixed, which may indicate some negligence, the number of comments and the number of modified test files suggests that the EH bugs are not less understood nether less tested.

While we observed reasonable results using simple word embeddings (TF-IDF and BoW), we believe there is still room to improve our results using more complex encoders, e.g., based on transformer models (DEVLIN *et al.*, 2019). Additionally, we could weigh the loss function to account for cases where correctly classifying some is more important than others. These weights can, e.g., be estimates of the (monetary) cost of miss-classifying a bug report. Methods to "handle" imbalance is a special case of this concept where the cost of getting a sample wrong is inversely proportional to the frequency of its label. We believe, however, that using these techniques can be miss-guiding outside the context where these costs are explicitly defined — and decided not to use "data balancing" procedures in our experiments.

6.1.2 Implications for Researchers

Our study brings at least two implications for researchers. The first one concerns the possibility of performing in-depth research on EH bugs using the proposed dataset to explore other dimensions such as reproducibility, testability, and the extent to which they impact other bugs and how they impact. The second implication is related to the first step to provide a labeled dataset to evaluate other ML and NLP techniques for the task of EH bugs classification.

6.1.3 Implications for Practitioners

The findings of our study show that the EH bugs take more time to be fixed them other kinds of bugs in our dataset. On the one hand, previous studies claim that EH bugs may cause severe consequences and must be quickly identified and fixed. On the other hand, based on our findings, these kinds of bugs are not receiving the expected prioritization. Perhaps developers could be taking time to identify the EH bugs. In this case, our approach to the automatic labeling of EH bugs could help developers be more aware of EH bugs.

6.2 Threats to Validity

The threats to the validity of our study are discussed using the classification presented by Wohlin *et al.* (2012). However, once we do not investigate causal relations, the internal validity was omitted.

6.2.1 Conclusion Validity

This threat affects the ability to draw correct conclusions about the relationship between treatment and outcome. To avoid this kind of threat, we carefully choose and employed (i) statistical methods to analyze the EH-Bug dataset; and (ii) different ML methods and NLP text encoding techniques to experimentally find the best combination for the task of automatic labeling of EH bugs.

6.2.2 Construct Validity

This threat refers to the extent to which the experiment setting reflects the theory. To avoid this threat we started from an existing dataset and employed strategies to assess the reliability of the manual labeling process (peer review, perspectives aligning, and agreement level analysis). Additionally, we tried to reproduce, using NLP techniques, the process used by developers to apply a label to a bug report (i.e., look at text fields to identify what kind of bug the report records).

6.2.3 External Validity

This threat limits the ability to generalize the results beyond the experiment setting. To alleviate this threat we did two actions: (i) we build a dataset from a long-lived real-world large-scale software project; and (ii) used strategies to make the ML models aware of some domain issues, such as data imbalance and EH-related keywords.

7 CONCLUSION AND FUTURE WORK

Exception handling (EH) is an error-recovery technique that allows developers to anticipate abnormal situations by implementing recovery actions. The way EH features are implemented in major programming languages leads developers to create different flows of control, reducing the overall debugging capability of the software, and presenting new challenges for software testing. Studies have reported that EH is often not well understood, poorly tested, and usually neglected. All these situations can lead to serious consequences such as system downtime, data loss, and security risk. Therefore, to avoid serious consequences, EH bugs must be quickly identified, prioritized, and assigned. However, this triage and labeling task depends mainly on the knowledge, time, and convenience of the bug reporter, which can lead to information reliability issues, requiring automation.

In this study, we empirically evaluated the idea of automatic labeling of EH bugs using ML and NLP techniques on features extracted from bug report fields. As a result, we obtained a hand-labeled dataset from an existing dataset containing 10 years of bug-fixing activity from the Apache Hadoop project resulting in 4516 bug reports with 943 (about 20%) of them labeled like EH bugs.

With the dataset of EH bugs obtained, we performed a controlled experiment combining four ML classifiers with two NLP strategies to extract features from the bug report text (Bag of Words and TF -IDF) and also evaluated whether the use of Bag of Words and TF-IDF only on keywords related to exception handling extracted from textual fields could improve the performance of ML models.

Our results show that the combination of NLP and ML techniques achieved ROC-AUC scores of up to 0.70 and recall ranging from 0.50 up to 0.62 for the automatic labeling of EH bugs. Additionally, considering only keywords related to EH as inputs, the ML models' performance was worst compared to using all words from the textual fields.

In future work, we plan to investigate how to build a model to perform the task of automatic maintainer assignment (i.e., who is the best fit to fix this EH bug?), evaluate both strategies (automatic labeling and assigning) in real settings and test different word embedding to improve the ML results.

The proposed dataset and all the code that support the findings of this study are available in Figshare with the identifier <https://doi.org/10.6084/m9.figshare.22735124.v1>.

BIBLIOGRAPHY

ASADUZZAMAN, M.; AHASANUZZAMAN, M.; ROY, C. K.; SCHNEIDER, K. A. How developers use exception handling in java? In: **Proceedings of the 13th International Conference on Mining Software Repositories**. New York, NY, USA: ACM, 2016. (MSR '16), p. 516–519. ISBN 978-1-4503-4186-8.

AUNG, T. W. W.; WAN, Y.; HUO, H.; SUI, Y. Multi-triage: A multi-task learning framework for bug triage. **Journal of Systems and Software**, v. 184, p. 111133, 2022. ISSN 0164-1212. Available at: https://www.sciencedirect.com/science/article/pii/S0164121221002302.

AVIZIENIS, A.; LAPRIE, J.-C.; RANDELL, B.; LANDWEHR, C. Basic concepts and taxonomy of dependable and secure computing. **IEEE Trans. Dependable Secur. Comput.**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 1, n. 1, p. 11–33, Jan. 2004. ISSN 1545-5971.

BARBOSA, E. A.; GARCIA, A.; BARBOS, S. D. J. Categorizing faults in exception handling: A study of open source projects. In: **Software Engineering (SBES), 2014 Brazilian Symposium on**. [S.l.: s.n.], 2014. p. 11–20.

CACHO, N.; BARBOSA, E. A.; ARAUJO, J.; PRANTO, F.; GARCIA, A.; CESAR, T.; SOARES, E.; CASSIO, A.; FILIPE, T.; GARCIA, I. How does exception handling behavior evolve? an exploratory study in java and c# applications. In: **Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution**. [S.l.]: IEEE, 2014. (ICSME'14), p. 31–40. ISBN 978-1-4799-6146-7.

CACHO, N.; CÉSAR, T.; FILIPE, T.; SOARES, E.; CASSIO, A.; SOUZA, R.; GARCIA, I.; BARBOSA, E. A.; GARCIA, A. Trading robustness for maintainability: An empirical study of evolving c# programs. In: **Proceedings of the 36th International Conference on Software Engineering**. [S.l.: s.n.], 2014. (ICSE 2014), p. 584–595. ISBN 978-1-4503-2756-5.

CATOLINO, G.; PALOMBA, F.; ZAIDMAN, A.; FERRUCCI, F. Not all bugs are the same: Understanding, characterizing, and classifying bug types. **Journal of Systems and Software**, v. 152, p. 165–181, 2019. ISSN 0164-1212. Available at: https://www.sciencedirect.com/science/article/pii/S0164121219300536.

CHANG, B.-M.; CHOI, K. A review on exception analysis. **Inf. Softw. Technol.**, Butterworth-Heinemann, Newton, MA, USA, v. 77, n. C, p. 1–16, Sep. 2016. ISSN 0950-5849.

CHAWLA, I.; SINGH, S. K. Automatic bug labeling using semantic information from lsi. In: **2014 Seventh International Conference on Contemporary Computing (IC3)**. [S.l.: s.n.], 2014. p. 376–381.

CHAWLA, I.; SINGH, S. K. An automated approach for bug categorization using fuzzy logic. In: **Proceedings of the 8th India Software Engineering Conference**. New York, NY, USA: Association for Computing Machinery, 2015. (ISEC '15), p. 90–99. ISBN 9781450334327. Available at: https://doi.org/10.1145/2723742.2723751.

CHEN, C.-T.; CHENG, Y. C.; HSIEH, C.-Y.; WU, I.-L. Exception handling refactorings: Directed by goals and driven by bug fixing. **J. Syst. Softw.**, Elsevier Science Inc., New York, NY, USA, v. 82, n. 2, p. 333–345, Feb. 2009. ISSN 0164-1212.

CHEN, H.; DOU, W.; JIANG, Y.; QIN, F. Understanding exception-related bugs in large-scale cloud systems. In: **2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)**. [S.l.: s.n.], 2019. p. 339–351.

CHEN, J.; HE, X.; LIN, Q.; ZHANG, H.; HAO, D.; GAO, F.; XU, Z.; DANG, Y.; ZHANG, D. Continuous incident triage for large-scale online service systems. In: **Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering**. IEEE Press, 2019. (ASE '19), p. 364–375. ISBN 9781728125084. Available at: https://doi.org/10.1109/ASE.2019.00042.

COELHO, R.; ALMEIDA, L.; GOUSIOS, G.; DEURSEN, A. V.; TREUDE, C. Exception handling bug hazards in android. **Empirical Softw. Engg.**, Kluwer Academic Publishers, Hingham, MA, USA, v. 22, n. 3, p. 1264–1304, Jun. 2017. ISSN 1382-3256.

COHEN, J. A coefficient of agreement for nominal scales. **Educational and Psychological Measurement**, v. 20, p. 37 – 46, 1960.

DALTON, F.; RIBEIRO, M.; PINTO, G.; FERNANDES, L.; GHEYI, R.; FONSECA, B. Is exceptional behavior testing an exception? an empirical assessment using java automated tests. In: **Proceedings of the Evaluation and Assessment in Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2020. (EASE '20), p. 170–179. ISBN 9781450377317. Available at: https://doi.org/10.1145/3383219.3383237.

DEVLIN, J.; CHANG, M.-W.; LEE, K.; TOUTANOVA, K. BERT: Pre-training of deep bidirectional transformers for language understanding. In: **Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)**. Minneapolis, Minnesota: [s.n.], 2019. p. 4171–4186.

EBERT, F.; CASTOR, F.; SEREBRENIK, A. An exploratory study on exception handling bugs in java programs. **J. Syst. Softw.**, Elsevier Science Inc., New York, NY, USA, v. 106, n. C, p. 82–101, Aug. 2015. ISSN 0164-1212.

EBERT, F.; CASTOR, F.; SEREBRENIK, A. A reflection on "an exploratory study on exception handling bugs in java programs". In: **2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)**. [S.l.: s.n.], 2020. p. 552–556.

ELZANATY, F.; REZK, C.; LIJBRINK, S.; BERGEN, W. van; COTE, M.; MCINTOSH, S. Automatic recovery of missing issue type labels. **IEEE Software**, v. 38, n. 3, p. 35–42, 2021.

FILHO, J. L. M.; ROCHA, L.; ANDRADE, R.; BRITTO, R. Preventing erosion in exception handling design using static-architecture conformance checking. In: **Proceedings of the 11th European Conference on Software Architecture**. Cham: Springer International Publishing, 2017. (ECSA '17), p. 67–83. ISBN 978-3-319-65831-5.

GALLARDO, R.; HOMMEL, S.; KANNAN, S.; GORDON, J.; ZAKHOUR, S. B. **The Java Tutorial: A Short Course on the Basics**. 6th. ed. [S.l.]: Addison-Wesley Professional, 2014. 864 p. (Java Series). ISBN 0134034082.

GARCIA, A. F.; RUBIRA, C. M.; ROMANOVSKY, A.; XU, J. A comparative study of exception handling mechanisms for building dependable object-oriented software. **Journal of Systems and Software**, v. 59, n. 2, p. 197–222, 2001. ISSN 0164-1212.

GOFFI, A.; GORLA, A.; ERNST, M. D.; PEZZè, M. Automatic generation of oracles for exceptional behaviors. In: **Proceedings of the 25th International Symposium on Software Testing and Analysis**. New York, NY, USA: ACM, 2016. (ISSTA 2016), p. 213–224. ISBN 978-1-4503-4390-9.

GOMES, L. A. F.; TORRES, R. da S.; CôRTES, M. L. Bug report severity level prediction in open source software: A survey and research opportunities. **Information and Software Technology**, v. 115, p. 58–78, 2019. ISSN 0950-5849. Available at: https://www.sciencedirect.com/science/article/pii/S0950584919301648.

GOODENOUGH, J. B. Exception handling: Issues and a proposed notation. **Communications of the ACM**, ACM Press, New York, NY, USA, v. 18, p. 683–696, December 1975. ISSN 0001-0782.

HU, H.; ZHANG, H.; XUAN, J.; SUN, W. Effective bug triage based on historical bug-fix information. In: **Proceedings of the 2014 IEEE 25th International Symposium on Software Reliability Engineering**. USA: IEEE Computer Society, 2014. (ISSRE '14), p. 122–132. ISBN 9781479960330. Available at: https://doi.org/10.1109/ISSRE.2014.17.

JONES, K. S. A statistical interpretation of term specificity and its application in retrieval. In: JOURNAL OF DOCUMENTATION. **Document Retrieval Systems**. GBR: Taylor Graham Publishing, 1988. p. 132–142. ISBN 0947568212.

KECHAGIA, M.; SPINELLIS, D. Undocumented and unchecked: Exceptions that spell trouble. In: **Proceedings of the 11th Working Conference on Mining Software Repositories**. New York, NY, USA: ACM, 2014. (MSR 2014), p. 312–315. ISBN 978-1-4503-2863-0.

KIENZLE, J. On exceptions and the software development life cycle. In: **Proceedings of the 4th International Workshop on Exception Handling**. New York, NY, USA: ACM Press, 2008. (WEH'08), p. 32–38. ISBN 978-1-60558-229-0.

KNUDSEN, J. Better exception-handling in block-structured systems. **IEEE Software**, v. 4, n. 3, p. 40–49, 1987.

KÖKSAL, O.; ÖZTÜRK, C. E. A survey on machine learning-based automated software bug report classification. In: **2022 International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)**. [S.l.: s.n.], 2022. p. 635–640.

LEE, S.-R.; HEO, M.-J.; LEE, C.-G.; KIM, M.; JEONG, G. Applying deep learning based automatic bug triager to industrial projects. In: **Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2017. (ESEC/FSE 2017), p. 926–931. ISBN 9781450351058. Available at: https://doi.org/10.1145/3106237.3117776.

LIMA, L. P.; ROCHA, L. S.; BEZERRA, C. I. M.; PAIXAO, M. Assessing exception handling testing practices in open-source libraries. **Empirical Software Engineering**, v. 26, n. 5, p. 85, Jun 2021. ISSN 1573-7616. Available at: https://doi.org/10.1007/s10664-021-09983-3.

MARCILIO, D.; FURIA, C. A. How java programmers test exceptional behavior. In: **2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)**. [S.l.: s.n.], 2021. p. 207–218.

MARINESCU, C. Are the classes that use exceptions defect prone? In: ACM. **Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution**. [S.l.], 2011. p. 56–60.

MARINESCU, C. Should we beware the exceptions? an empirical study on the eclipse project. In: IEEE. **Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2013 15th International Symposium on**. [S.l.], 2013. p. 250–257.

MELO, H.; COELHO, R.; TREUDE, C. Unveiling exception handling guidelines adopted by java developers. In: **2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)**. [S.l.: s.n.], 2019. p. 128–139.

MILLER, R.; TRIPATHI, A. Issues with exception handling in object-oriented systems. In: AKSIT, M.; MATSUOKA, S. (Ed.). **ECOOP'97 - Object-Oriented Programming**. [S.l.]: Springer Berlin / Heidelberg, 1997, (Lecture Notes in Computer Science, v. 1241). p. 85–103.

PáDUA, G. B. de; SHANG, W. Studying the prevalence of exception handling anti-patterns. In: **Proceedings of the 25th International Conference on Program Comprehension**. Piscataway, NJ, USA: IEEE Press, 2017. (ICPC '17), p. 328–331. ISBN 978-1-5386-0535-6.

PáDUA, G. B. de; SHANG, W. Studying the relationship between exception handling practices and post-release defects. In: **Proceedings of the 15th International Conference on Mining Software Repositories**. New York, NY, USA: ACM, 2018. (MSR '18), p. 564–575. ISBN 978-1-4503-5716-6. Available at: http://doi.acm.org/10.1145/3196398.3196435.

PANDEY, N.; SANYAL, D. K.; HUDAIT, A.; SEN, A. Automated classification of software issue reports using machine learning techniques: An empirical study. **Innov. Syst. Softw. Eng.**, Springer-Verlag, Berlin, Heidelberg, v. 13, n. 4, p. 279–297, dec 2017. ISSN 1614-5046. Available at: https://doi.org/10.1007/s11334-017-0294-1.

PARNAS, D. L.; WüRGES, H. Response to undesired events in software systems. In: **Proceedings of the 2nd International Conference on Software Engineering**. Los Alamitos, CA, USA: IEEE Computer Society Press, 1976. (ICSE'76), p. 437–446.

PETERS, F.; TUN, T. T.; YU, Y.; NUSEIBEH, B. Text filtering and ranking for security bug report prediction. **IEEE Transactions on Software Engineering**, v. 45, n. 6, p. 615–631, 2019.

PICUS, O.; SERBAN, C. Bugsby: A tool support for bug triage automation. In: **Proceedings of the 2nd ACM International Workshop on AI and Software Testing/Analysis**. New York, NY, USA: Association for Computing Machinery, 2022. (AISTA 2022), p. 17–20. ISBN 9781450393874. Available at: https://doi.org/10.1145/3536168.3543301.

ROBILLARD, M. P.; MURPHY, G. C. Static analysis to support the evolution of exception structure in object-oriented systems. **ACM Trans. Softw. Eng. Methodol.**, ACM, New York, NY, USA, v. 12, n. 2, p. 191–221, Apr. 2003. ISSN 1049-331X. Available at: http://doi.acm.org/10.1145/941566.941569.

SAWADPONG, P.; ALLEN, E. B. Software defect prediction using exception handling call graphs: A case study. In: IEEE. **High Assurance Systems Engineering (HASE), 2016 IEEE 17th International Symposium on**. [S.l.], 2016. p. 55–62.

SAWADPONG, P.; ALLEN, E. B.; WILLIAMS, B. J. Exception handling defects: An empirical study. In: IEEE. **High-Assurance Systems Engineering (HASE), 2012 IEEE 14th International Symposium on**. [S.l.], 2012. p. 90–97.

SHAH, H.; GORG, C.; HARROLD, M. J. Understanding exception handling: Viewpoints of novices and experts. **IEEE Trans. Softw. Eng.**, IEEE Press, Piscataway, NJ, USA, v. 36, n. 2, p. 150–161, Mar. 2010. ISSN 0098-5589.

SHAHROKNI, A.; FELDT, R. A systematic review of software robustness. **Inf. Softw. Technol.**, Butterworth-Heinemann, Newton, MA, USA, v. 55, n. 1, p. 1–17, Jan. 2013. ISSN 0950-5849.

SINHA, S.; HARROLD, M. J. Analysis and testing of programs with exception handling constructs. **IEEE Transactions on Software Engineering**, v. 26, n. 9, p. 849–871, Sept 2000. ISSN 0098-5589.

SOUSA, D. B. C. de; MAIA, P. H. M.; ROCHA, L. S.; VIANA, W. Studying the evolution of exception handling anti-patterns in a long-lived large-scale project. **Journal of the Brazilian Computer Society**, v. 26, n. 1, p. 1, Jan 2020. ISSN 1678-4804. Available at: https://doi.org/10.1186/s13173-019-0095-5.

SPADINI, D.; ANICHE, M.; BACCHELLI, A. PyDriller: Python framework for mining software repositories. In: **Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018**. New York, New York, USA: ACM Press, 2018. p. 908–911. ISBN 9781450355735. Available at: http://dl.acm.org/citation.cfm?doid=3236024.3264598.

TIAN, Y.; LO, D.; XIA, X.; SUN, C. Automated prediction of bug report priority using multi-factor analysis. **Empirical Software Engineering**, v. 20, n. 5, p. 1354–1383, Oct 2015. ISSN 1573-7616. Available at: https://doi.org/10.1007/s10664-014-9331-y.

UDDIN, J.; GHAZALI, R.; DERIS, M. M.; NASEEM, R.; SHAH, H. A survey on bug prioritization. **Artif. Intell. Rev.**, Kluwer Academic Publishers, USA, v. 47, n. 2, p. 145–180, feb 2017. ISSN 0269-2821. Available at: https://doi.org/10.1007/s10462-016-9478-6.

VIEIRA, R.; SILVA, A. da; ROCHA, L.; GOMES, J. a. P. From reports to bug-fix commits: A 10 years dataset of bug-fixing activity from 55 apache's open source projects. In: **Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2019. (PROMISE'19), p. 80–89. ISBN 9781450372336. Available at: https://doi.org/10.1145/3345629.3345639.

VIEIRA, R. G.; MATTOS, C. L. C.; ROCHA, L. S.; GOMES, J. P. P.; PAIXÃO, M. The role of bug report evolution in reliable fixing estimation. **Empirical Software Engineering**, v. 27, n. 7, p. 164, Sep 2022. ISSN 1573-7616. Available at: https://doi.org/10.1007/s10664-022-10213-7.

WHITE, T. **Hadoop: The Definitive Guide: Storage and Analysis at Internet Scale 4th Edition**. [S.l.]: O'Reilly Media, 2015. ISBN 9781491901632.

WOHLIN, C.; RUNESON, P.; HST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLN, A. **Experimentation in Software Engineering**. [S.l.]: Springer Publishing Company, Incorporated, 2012. ISBN 3642290434.

XU, B.; HUANG, J. Z.; WILLIAMS, G.; WANG, Q.; YE, Y. Classifying very high-dimensional data with random forests built from small subspaces. **Int. J. Data Warehous. Min.**, v. 8, n. 2, p. 44–63, apr 2012.

ZHANG, J.; WANG, X.; ZHANG, H.; SUN, H.; PU, Y.; LIU, X. Learning to handle exceptions. In: **Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2021. (ASE '20), p. 29–41. ISBN 9781450367684. Available at: https://doi.org/10.1145/3324884.3416568.

ZHANG, P.; ELBAUM, S. Amplifying tests to validate exception handling code: An extended study in the mobile application domain. **ACM Trans. Softw. Eng. Methodol.**, ACM, New York, NY, USA, v. 23, n. 4, p. 32:1–32:28, Sep. 2014. ISSN 1049-331X.