



Towards automatic labeling of exception handling bugs: A case study of 10 years bug-fixing in Apache Hadoop

Antônio José A. da Silva¹ · Renan G. Vieira¹ · Diego P. P. Mesquita² ·
João Paulo P. Gomes¹ · Lincoln S. Rocha¹

Accepted: 2 May 2024 / Published online: 5 June 2024

© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2024

Abstract

Context Exception handling (EH) bugs stem from incorrect usage of exception handling mechanisms (EHMs) 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 AI-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 to evaluate 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 to provide 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. We also labeled 2,584 non-EH bugs analyzing their bug-fixing code and creating a dataset composed of 7,100 bug reports. Then, we used word embedding techniques (Bag-of-Words and TF-IDF) to summarize the textual fields of bug reports. Subsequently, we used these embeddings to fit five classes of ML methods and evaluate them on unseen data. We also evaluated a pre-trained transformer-based model using the complete textual fields. We have also evaluated whether considering only EH keywords is enough to achieve high predictive performance.

Results Our results show that using a pre-trained DistilBERT with a linear layer trained with our proposed dataset can reasonably label EH bugs, achieving ROC-AUC scores of up to 0.88. The combination of NLP and ML traditional techniques achieved ROC-AUC scores of up to 0.74 and recall up to 0.56. As a sanity check, we also evaluate methods using embeddings extracted solely from keywords. Considering ROC-AUC as the primary concern, for the majority of ML methods tested, the analysis suggests that keywords alone are not sufficient to characterize reports of EH bugs, although this can change based on other metrics (such as recall and precision) or ML methods (e.g., Random Forest).

Communicated by: Weiyi (Ian) Shang and Gema Rodríguez-Pérez

Extended author information available on the last page of the article

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 use of ML techniques, specially transformer-base models, 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

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 (Shahrokhni and 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, b). 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 and Murphy 2003; Chang and Choi 2016) and posing new challenges to software testing (Sinha and Harrold 2000; Zhang and Elbaum 2014; Dalton et al. 2020; Marcilio and 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 and Spinellis 2014; Zhang and Elbaum 2014; Asaduzzaman et al. 2016; Goffi et al. 2016; Chang and 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 EHM 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 and Serban 2022; Köksal and 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; de Pádua and Shang 2017; Ebert et al. 2020; de Sousa et al. 2020) and investigating the existence of statistical relationships between them (Marinescu 2011; Sawadpong et al. 2012; Marinescu 2013; Sawadpong and Allen 2016; de Pádua and 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, several studies 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 and Singh 2015; Aung et al. 2022), labeling the kind of bug (e.g., security and permission) (Chawla and Singh 2014; Peters et al. 2019; Catolino et al. 2019; Elzanaty et al. 2021), assigning bug severity (Gomes et al. 2019; Picus and 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.

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. Additionally, we also labeled 2,584 non-EH bugs analyzing their bug-fixing code, and creating a dataset composed of 7,100 bug reports. 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 concerning 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 six ML classifiers (Support Vector Classifier, Multinomial Naive Bayes, Linear Regression, Random Forest, AdaBoost Classifier and pre-trained DistilBERT) 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 using a pre-trained DistilBERT with a classification linear layer trained with our proposed dataset can reasonably label EH bugs, achieving ROC-AUC scores of up to 0.88. The combination of NLP and ML traditional techniques achieved ROC-AUC scores of up to 0.74. Additionally, considering only keywords related to EH and being AUC the major concern, the ML models' performance was worst when compared to the ones that use the full text (DistilBERT) and all keywords in the textual fields. 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 to a 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 developers' 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 and Würges 1976). Exception handling provides a means to structure fault tolerance activities through error recovery (Garcia et al. 2001). Additionally, exceptions can model other situations (Miller and 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 and represent exceptional conditions that a robust application should anticipate and recover from. Unchecked exceptions are those that inherit, directly or indirectly, from `Error` or `RuntimeException` classes (both from `java.lang` package) and 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`



Hadoop HDFS / HDFS-13100

Handle `IllegalArgumentException` when `GETSERVERDEFAULTS` is not implemented in `webhdfs`.

Fig. 1 Summary content of HDFS-13100 bug report

blocks. Each `catch` block catches a specific exception type and encloses the exception handler code. The `finally` block is optional, but if 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-13100¹ from Hadoop’s HDFS module (see Fig. 1). In fact, the cause of HDFS-13100 bug is the incorrect handling of two exceptions: `UnsupportedOperationException` and `IllegalArgumentException`. The fix action addresses the bug by implementing the following rules: (i) if the required operation is not supported, the `UnsupportedOperationException` must be thrown; and (ii) if the given parameter is not a legal one, the exception `IllegalArgumentException` must be thrown.

In other cases, it is necessary to go beyond and also inspect the bug report description, as in the case of MAPREDUCE-6156² bug report of the Hadoop’s MapReduce module (see Fig. 2). The description of MAPREDUCE-6156 bug gives us the idea that the handler (catch block) associated with the `IOException` does not deal 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-1505³ bug report of Hadoop’s HDFS module is an example of that (see Fig. 3). It is possible to infer from the comments that the maintainers reached an understanding that the

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

```

Fig. 2 Description of MAPREDUCE-6156 bug report

cause of the reported bug is the lack of throwing an exception to characterize the failure to save in all image directories.

3 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 3.1) and then we describe the EH-Bug dataset itself (Section 3.2).

3.1 The Original Dataset

Vieira et al. (2019) proposed a dataset comprising a set of 10-year 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 Jira⁴ and Git⁵ 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 Jira⁶ library to automate the mining process.

⁴ <https://issues.apache.org/jira>

⁵ <http://gitbox.apache.org>

⁶ <https://jira.readthedocs.io/>




- ▼  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.
-

Fig. 3 Comments of HDFS-1505 bug report

Second, they used the bug report ID of mined issues from Jira to mine Git repository using Pydriller⁷ (Spadini et al. 2018) framework to retrieve the respective fixing commits, resulting in the first dataset they called `snapshot`.

Using the list of retrieved issue 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 `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 NLTK⁸, 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 (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

⁷ <https://github.com/ishepard/pydriller>

⁸ <https://www.nltk.org/>

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
	Source (15)	NonSrcAddFiles
NonSrcDelFiles		
NonSrcModFiles		

Table 1 continued

From	Type	Field
		NonSrcAddLines
		NonSrcDelLines
		SrcAddFiles
		SrcDelFiles
		SrcModFiles
		SrcAddLines
		SrcDelLines
		TestAddFiles
		TestDelFiles
		TestModFiles
		TestAddLines
		TestDelLines

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

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 they were mined from Jira; and (iii) `commit-log`: A number of bug reports are related to some commit that fixes that bug. This dataset stores commit information related to each report that has one. The dataset entries bring detailed information about each file modified by bug-fix commits. The data fields are shown in Table 4.

3.2 Our Dataset

Our EH-Bug dataset was derived from Vieira et al. (2019) dataset (see Section 3.1) considering only the Apache Hadoop project. Hadoop is an open-source framework developed by 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) Distributed Filesystem (HDFS): a distributed filesystem that runs on clusters designed for storing very large files and providing high-throughput access; (iv) YARN (Yet Another Resource Negotiator): a framework for job scheduling/monitoring and cluster

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

resource management. It 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 the number of bugs for each component considering both filtering steps we explain later.

Hadoop (and consequently its components) was 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 a 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 Fig. 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 second filtering results in **4,516** bug reports.

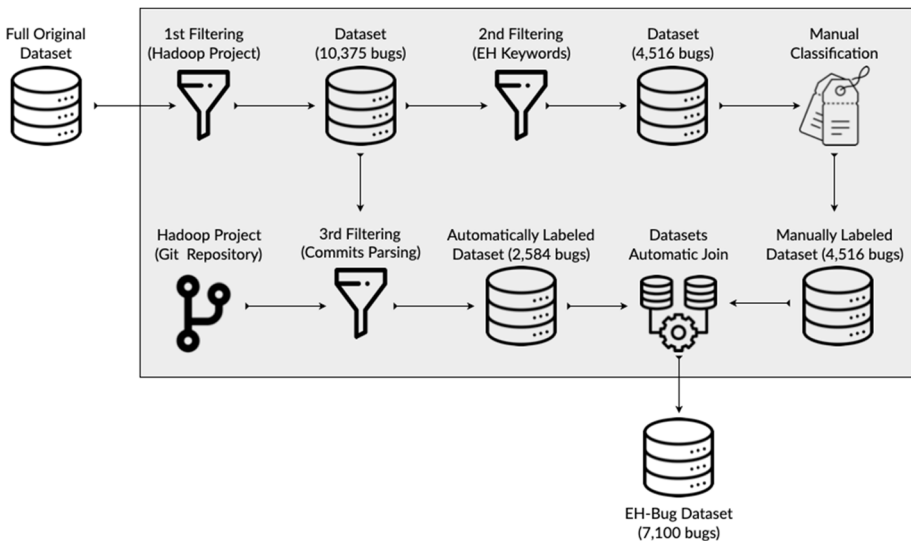


Fig. 4 Dataset creation methodology flow covering all the steps to the final EH-Bug dataset

All bug reports resulting from the second filtering were manually inspected and classified into two categories: EH bug and non-EH bug. The classification was based on the text in summary, description, and comments fields, using the definition presented in Section 2.3. We also look for exception-handling code updates in the bug-fixing patch (when available in the bug report) to assert even more our classification. 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 (a senior software engineer with large experience in software reliability and robustness tests) 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.

We also perform a 3rd filtering trying to analyze the bug reports that don’t match the second filtering criteria. In this case, we select only the bug reports that do not have any EH-related keywords in their text fields (summary, description, comments, and commits message). We used the `commit-log` dataset from the original dataset to inspect only bug reports that have fixing commits linked to them. Therefore, we mine each fixing commit and automatically parser each `.java` file changed in the commit and check if the changed parts affect the EH code, i.e., if the changes take place within a `try`, `catch`, or `finally` block or contains the `throw` or `throws` statements. Note that if a fixing commit affects the EH code, it is not possible to say that the linked bug is or is not an EH bug. However, if all fixing commits of a bug do not affect the EH code, it is reasonable to assume that this bug is a non-EH bug. Thus, our third filtering resulted in a total of 2,584 non-EH bugs. Finally, we joined both automatically and manually labeled datasets, resulting in a final EH-Bug dataset with 7,100 bugs.

After the first manual classification, we performed an evaluation to assess the classification reliability (see Fig. 5). With this purpose, we performed a second manual classification and computed the level of agreement between both sets of labels. The second manual classification was performed by two other independent authors of this study on a randomly selected significantly-sized 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. As a final validation step, the second and third labelers discuss their discordances and reach an agreement, generating the final labels for the selected sample dataset.

We used Cohen’s Kappa coefficient (Cohen 1960) to measure the level of agreement between the two labelings (the sample labeled by the first labeler and the sample agreed between the second and third 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.

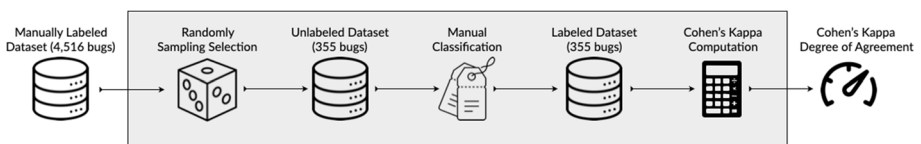


Fig. 5 Manual dataset labeling evaluation methodology flow

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 0.876. Thus, according to Table 6, Cohen's Kappa score obtained is classified as almost perfect of agreement, being a result considered relevant to guarantee the reliability of our dataset.

4 The EH-Bug 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 discussion.

4.1 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 and Spinellis 2014; Zhang and Elbaum 2014; Asaduzzaman et al. 2016; Goffi et al. 2016; Chang and 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. In this analysis, we group all the bugs not classified in the labeling phase as EH bugs, not distinguishing them as any specific type of bug. 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 (16 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 3.1. 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 three standard deviations from the sample mean. Removing outliers based on the sample standard deviation is a common technique, and we select the outlier factor of three to be very conservative once values as one or two can significantly reduce the sample size. In this case, only 136 (0.02%) of the 6283 reports that remained from the previous filters were removed.

4.2 Results

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

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

Summary of RQ1: The EH bugs are significantly (i) more prioritized with a negligible effect size and (ii) take more time to be fixed with a small effect size 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.

The first group of hypotheses contains the null hypothesis (\mathcal{H}_0^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 (\mathcal{H}_1^A) or a lower (\mathcal{H}_2^A) level of priority than non-EH bugs.

The second group of hypotheses contains the null hypothesis (\mathcal{H}_0^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 (\mathcal{H}_1^B) or a lower (\mathcal{H}_2^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

Table 7 Summary of hypothesis statement, the statistics test, and the Cohen’s Delta effect size results

MW hypothesis	p-value	Effect size
\mathcal{H}_0^A : EH_PRIORITY = NON_EH_PRIORITY (X)	2.338×10^{-3}	0.0904
\mathcal{H}_1^A : EH_PRIORITY > NON_EH_PRIORITY (✓)		(negligible)
\mathcal{H}_2^A : EH_PRIORITY < NON_EH_PRIORITY (X)		
\mathcal{H}_0^B : EH_FIXING-TIME = NON_EH_FIXING-TIME (X)	8.912×10^{-16}	0.2213
\mathcal{H}_1^B : EH_FIXING-TIME > NON_EH_FIXING-TIME (✓)		(small)
\mathcal{H}_2^B : EH_FIXING-TIME < NON_EH_FIXING-TIME (X)		
\mathcal{H}_0^C : EH_COMMENTS = NON_EH_COMMENTS (X)	6.274×10^{-25}	0.3994
\mathcal{H}_1^C : EH_COMMENTS > NON_EH_COMMENTS (✓)		(medium)
\mathcal{H}_2^C : EH_COMMENTS < NON_EH_COMMENTS (X)		
\mathcal{H}_0^D : EH_TEST = NON_EH_TEST (X)	3.342×10^{-24}	0.2899
\mathcal{H}_1^D : EH_TEST > NON_EH_TEST (✓)		(small)
\mathcal{H}_2^D : EH_TEST < NON_EH_TEST (X)		

The symbols ✓ and X indicate the result of the null hypothesis test (✓ fail to reject, and X reject). The Cohen’s Delta effect size interpretation: negligible = [0, 0.147), small = [0.147, 0.33), medium = [0.33, 0.474), and large = [0.474, 1]

Table 8 Bug report priority classification in Jira platform

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.

Table 9 Distribution of EH and non-EH bugs priority

Priority	EH Bugs		Non-EH Bugs	
	Number	(%)	Number	(%)
Blocker	59	07.42	539	09.83
Critical	117	14.70	567	10.34
Major	482	60.56	3106	56.60
Minor	119	14.94	982	17.89
Trivial	19	02.38	293	05.34

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	38.81	0.00	621.70	6.67	86.92

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 (7.42%) when compared with non-EH bugs (9.83%). Additionally, it is possible to see that EH bugs tend to have a lower 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 \mathcal{H}_0^A , accepting the alternative hypothesis \mathcal{H}_1^A . This indicates that not only the priority of EH and non-EH bugs are statistically different but also that the EH bugs are statistically more prioritized than EH bugs but with a negligible effect size. In fact, the results shows that the average priority of EH bugs is 0.0904 standard deviations higher than the average priority of non-EH bugs.

Table 10 presents the bug-fixing time descriptive statistics for EH and non-EH bugs, while Fig. 6 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, the mean and median 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 \mathcal{H}_0^B and accept \mathcal{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 with a small effect size. The results show that the average bug-fixing time of EH bugs is 0.2213 standard deviations higher than the average bug-fixing time of non-EH bugs.

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

Summary of RQ2: The EH bugs are significantly more discussed with a medium effect size 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 (\mathcal{H}_0^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

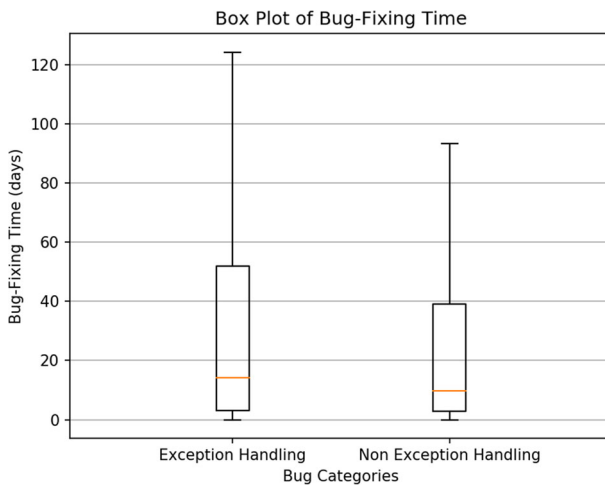


Fig. 6 Bug-fixing time boxplot. When comparing the boxplots, it is possible to see that the interquartile EH bug-fixing time is larger than the non-EH bug-fixing time

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	19.80	4	55	17	10.78
Non-EH	15.87	2	55	14	9.36

EH bugs have either a higher (\mathcal{H}_1^C) or a lower (\mathcal{H}_2^C) number of comments compared to non-EH bugs.

Table 11 presents the descriptive statistics for the number of comments posted in EH and non-EH bug reports, while Fig. 7 shows the boxplot of the number of comments. Upon comparing the boxplots and statistics, it is evident that they are very similar, but the EH bugs present slightly higher values. This observation is confirmed by the statistical test results that reject the null hypothesis \mathcal{H}_0^C and accept \mathcal{H}_1^C . Therefore, we can assume that the number of comments in EH bug reports is statistically higher than non-EH bug reports with a medium size effect of 0.3994.

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

Summary of RQ3: The EH bugs are significantly more tested with a small effect size 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 (\mathcal{H}_0^D) stating that there is no difference in the number of changed test files between EH and non-EH bugs. The alternative hypotheses, on the other hand, assume that EH bugs have either a higher (\mathcal{H}_1^D) or a lower (\mathcal{H}_2^D) number of changed test files than non-EH bugs.

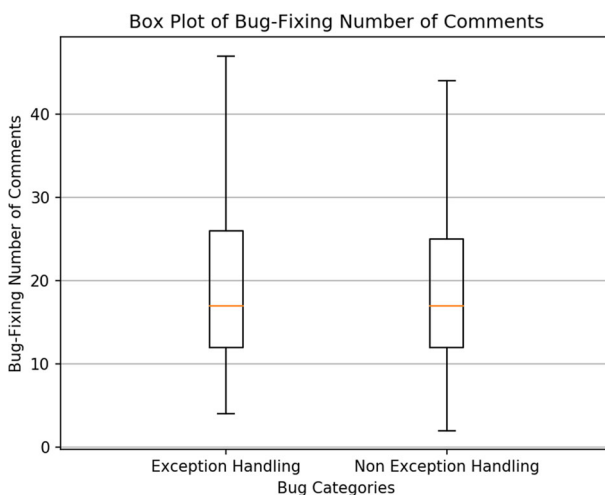


Fig. 7 Number of comments boxplot. When comparing the boxplots, it is possible to see that EH bugs tend to present a higher number of comments than non-EH bugs

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	1.086	0	10	1	1.32
Non-EH	0.742	0	10	0	1.16

Table 12 presents descriptive statistics for the number of test files changed in fixing commits of both EH and non-EH bugs. Meanwhile, Fig. 8 displays a boxplot of the number of test files changed. Upon comparing the boxplots and statistics, it is evident that they are similar, but the EH bugs present slightly higher values. This observation is confirmed by the statistical test results that reject the null hypothesis \mathcal{H}_0^D and accept \mathcal{H}_1^D . Therefore, we can assume that the number of test files changed in fixing commits of EH bugs is significantly higher but with a small effect size of 0.2899.

4.3 Discussion

The main goal of this analysis was to investigate whether the EH bugs receive less attention from developers compared with non-EH bugs in bug-fixing tasks. Therefore, we observed that for two dimensions (priority and fixing time) present EH bugs as being more prioritized and demanding more time to be fixed. However, it is important to notice that although the results for **RQ1** concerning priority and bug-fixing time reached a statistically significant p-value, the associated effect size suggests caution in the interpretation of the findings on this matter.

When we looked at the other two dimensions (discussion and testing) we observed that EH bugs are more discussed (**RQ2**) and more tested (**RQ3**). Considering all the results, it is more reasonable to think of EH bugs being more complex (as being more prioritized, taking more

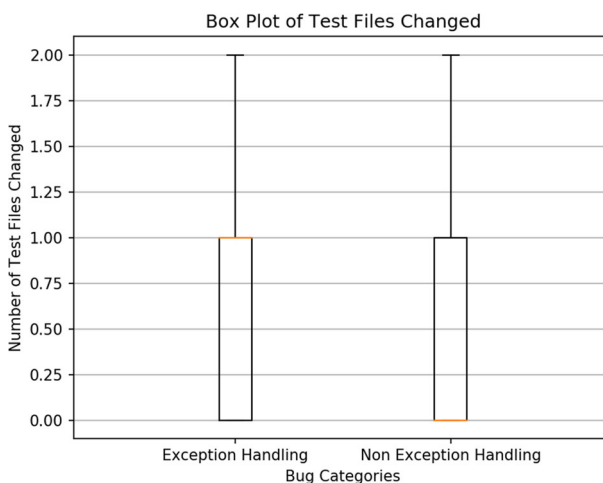


Fig. 8 Boxplot of the number of test files changed in fixing commits. Upon comparing the boxplot, it is evident that they are very similar

time to be fixed, more discussed, and demanding more changes in text files) than neglected. Nonetheless, it is also worth mentioning that the associated effect size, especially for the number of test files changed, suggests caution in interpreting the findings.

5 The EH-Bug Classification Model

In this section, 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. Despite 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 six models - Support Vector Classifier (SVC), Multinomial Naive Bayes (MNB), Linear Regression (LRC), Random Forest (RFC), AdaBoost Classifier (ABC), and DistilBERT (DBERT) (Sanh et al. 2019), a light pre-trained version of BERT, the transformer-based model proposed by Google - 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) (Sparck Jones 1988) (for the DBERT we use the full textual fields with no processing to train a classification layer). We also evaluate two different sets of words: i) All 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 21 results based on the combination of five models, two different NLP encodings, and two sets of features ($5 \times 2 \times 2 = 20$) (plus the use of integral fields with DistilBERT). We also define a simple rule-based classifier **baseline** to gauge the EH bug classification model performance: if the report has any of the EHK in summary, description, or comments, then it is EH bug; otherwise, it is a non-EH bug.

We train the models using 5-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

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'};
Support Vector Classifier	solver: {'liblinear', 'lbfgs'};
Random Forest	C: [0.01, 0.05, 0.1, 0.5, 1, 5, 10] n_estimators: [5, 30, 50, 75, 100, 150, 200]
AdaBoost Classifier	max_depth: [4, 5, 6, 7, 8, None] n_estimators: [5, 30, 50, 75, 100, 150, 200]
DistilBERT	learning_rate=5e-5, epochs=3, eight_decay=0.01, warmup_steps=100

of the scikit-learn package for the parameters and suggest the official documentation⁹ for more details about their meaning.

5.3 Results

Table 14 presents the average and standard deviation values for several metrics of the 5-fold runs (except for the baseline, where we show the classifier’s performance in the whole dataset). We highlight the best results of each metric in **boldface**.

First, we highlight the baseline lower metric values, which justify using some machine learning approach. Notwithstanding the high recall value - indicating that all EH bugs have at least one of the exception-handling keywords - the low precision suggests that the rule-based baseline produces many false positive classifications. Considering all the results, we highlight the results obtained by using DBERT. The transformer-based model provides the best accuracy, F1 and ROC-AUC results, with values of 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 DBERT for presenting the best balance between recall and precision compared to the other results, while obtaining the highest values of accuracy, ROC-AUC and F1-measure.

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, LRC, and ABC) and an NLP word embedding (BoW or TF-IDF), we compare the

⁹ <https://scikit-learn.org/stable/modules/classes.html>

Table 14 The EH models for classification results

NLP (Tokens)	Models	Precision	Recall	Accuracy	ROC AUC	F1
BoW (EHK)	Baseline	0.21	1.00	0.21	–	0.35
BoW	MNB	0.26 ± 0.0	0.72 ± 0.0	0.69 ± 0.0	0.70 ± 0.0	0.38 ± 0.0
	LRC	0.53 ± 0.0	0.56 ± 0.0	0.87 ± 0.0	0.74 ± 0.0	0.54 ± 0.0
	RFC	0.46 ± 0.1	0.18 ± 0.0	0.86 ± 0.0	0.57 ± 0.0	0.26 ± 0.1
	SVC	0.51 ± 0.1	0.53 ± 0.1	0.86 ± 0.0	0.72 ± 0.0	0.51 ± 0.1
	ABC	0.68 ± 0.0	0.48 ± 0.0	0.90 ± 0.0	0.72 ± 0.0	0.56 ± 0.0
TF-IDF	MNB	0.63 ± 0.1	0.07 ± 0.0	0.87 ± 0.0	0.53 ± 0.0	0.12 ± 0.1
	LRC	0.66 ± 0.0	0.40 ± 0.1	0.89 ± 0.0	0.68 ± 0.0	0.50 ± 0.1
	RFC	0.46 ± 0.1	0.14 ± 0.0	0.86 ± 0.0	0.55 ± 0.0	0.21 ± 0.0
	SVC	0.76 ± 0.1	0.27 ± 0.0	0.89 ± 0.0	0.63 ± 0.0	0.40 ± 0.1
	ABC	0.65 ± 0.0	0.49 ± 0.0	0.89 ± 0.0	0.72 ± 0.0	0.56 ± 0.0
BoW (EH)	MNB	0.56 ± 0.1	0.50 ± 0.1	0.88 ± 0.0	0.72 ± 0.0	0.53 ± 0.1
	LRC	0.71 ± 0.1	0.35 ± 0.0	0.89 ± 0.0	0.66 ± 0.0	0.46 ± 0.1
	RFC	0.62 ± 0.1	0.45 ± 0.1	0.89 ± 0.0	0.70 ± 0.0	0.52 ± 0.1
	SVC	0.71 ± 0.1	0.33 ± 0.1	0.89 ± 0.0	0.65 ± 0.0	0.45 ± 0.1
	ABC	0.67 ± 0.0	0.43 ± 0.1	0.89 ± 0.0	0.70 ± 0.0	0.52 ± 0.0
TF-IDF (EH)	MNB	0.00 ± 0.0	0.00 ± 0.0	0.86 ± 0.0	0.50 ± 0.0	0.00 ± 0.0
	LRC	0.69 ± 0.1	0.28 ± 0.0	0.88 ± 0.0	0.63 ± 0.0	0.40 ± 0.1
	RFC	0.60 ± 0.1	0.42 ± 0.0	0.88 ± 0.0	0.69 ± 0.0	0.50 ± 0.0
	SVC	0.73 ± 0.1	0.20 ± 0.0	0.88 ± 0.0	0.59 ± 0.0	0.31 ± 0.0
	ABC	0.66 ± 0.0	0.46 ± 0.0	0.89 ± 0.0	0.71 ± 0.0	0.54 ± 0.0
Full Text	DBERT	0.66 ± 0.0	0.54 ± 0.0	0.90 ± 0.0	0.88 ± 0.0	0.59 ± 0.0

ROC-AUC performance in each test fold obtained using EHK and AW — subtracting the first from the latter. We repeat the experiment 5 times (using 5-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 9 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).

5.4 Discussion

While we observed reasonable results using simple word embeddings (TF-IDF and BoW), the best results were obtained by using the full text to train a classification layer on the pre-trained DistilBERT transformer-based model. The recall values are around 0.54, and none of the other classifiers can provide a higher recall value while maintaining acceptable metrics values, indicating that there is still room for improvement. As the transformers model the state of the art to deal with textual data, we believe that the next step would be to label more EH bug reports, as most of the data is composed of non-EH bugs. Additionally, we could weigh

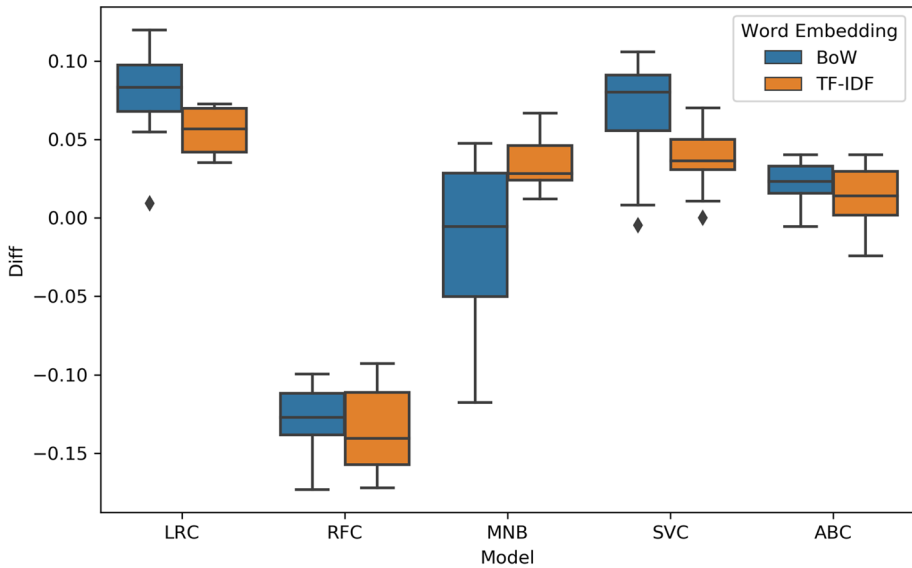


Fig. 9 Effect measure on ROC-AUC of using $AW \times EHK$ on five models

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 are 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 misleading outside the context where these costs are explicitly defined — and decided not to use “data balancing” procedures in our experiments.

6 Overall Discussion and Implications

In this section, we discuss our results (Section 6.1) and their implications for both, researchers (Section 6.2) and practitioners (Section 6.3).

6.1 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 and Choi 2016; Filho et al. 2017). With the dataset we may verify if this is not necessarily reflected in the bug-fixing process. Our results indicate that EH bug reports, in a small amount, are more prioritized and demand more time to be fixed, which the latter may indicate some negligence or some higher level of complexity that demands more effort to be fixed. The number of comments and modified test files of EH bugs being higher than non-EH bugs

suggests that they demand more discussion and more testing, which both can be seen as an indication of complexity. However, all those dimensions must be considered proxy variables to the complexity and only be seen as a hypothesis to explain those differences between the groups of EH bugs and non-EH bugs. This is aggravated by the fact that only the comments hypothesis test presented a medium effect size, while fixing-time, priority, and test files presented negligible and small ones.

In our case study, we only use the Hadoop components as the source and its main programming language is Java. In our investigations, we evaluate if only using Java EH keywords is enough to classify a bug report as an EH bug or if adding more textual context (keywords unrelated to the EH Java keywords) provides better results. Our experiments indicate that other words (coded by the NLP embeddings) plus the EH keywords lead to better results. Thus, up to a certain limit, our classification model could be useful to label EH bugs in other programming languages, such as C#, C++, and Python.

Our proposal may help the triage process, which is typically done by reading each bug report to understand its nature better. Right after the report opens, the automatic labeler should be able to identify the EH bugs. At this point, the label will be more helpful to the bug triage process. However, marking a report as an EH bug after the triage process (in cases where this classification is not reliable in the initial version of the report) is also beneficial once it can be used for subsequent analysis and documentation purposes. Finally, evaluating how practitioners perceive the automatic labeling tools in the bug-fixing process is mandatory, especially in the triage process. It is necessary to map what type of category and how this labeling helps them to ensure their software quality in future research.

6.2 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.

6.3 Implications for Practitioners

The findings of our study show that the EH bugs take more time to be fixed than other kinds of bugs in our dataset, even if they tend to be slightly more prioritized. 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 scheduling prioritization. Perhaps developers could be taking time to start engaging in the fixing of EH bugs. In this case, our approach to the automatic labeling of EH bugs could be used as a plugin of a bug tracking tool to help developers be more aware and prioritize the fixing of EH bugs.

7 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 did not investigate causal relations, the internal validity was omitted.

7.1 Conclusion Validity

Threats to the conclusion validity are concerned with factors that impact the capacity to make accurate inferences about the relationship between the treatment and the outcome of an experiment. To avoid this kind of threat, we carefully chose and employed (i) well-known statistical method (Mann-Whitney) to analyze the EH-Bug dataset, observing its assumptions (e.g., samples distribution, dependence, and size), trying to avoid wrong conclusions; 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 using well-established performance metrics as proxies (e.g., ROC AUC and F1).

7.2 Construct Validity

Threats to the construct validity concerns generalizing the result of the research to the concept or theory behind the study. To avoid this threat, we employed a peer debriefing strategy to validate the research design and document review. The aim was to prevent discrepancies in result interpretation. Furthermore, 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., looking at text fields to identify what kind of bug the report records).

7.3 External Validity

This threat limits the ability to generalize the results beyond the experiment setting. The work's findings were based on a case study of one single project: Hadoop. Therefore, the results can not be generalized for other projects, especially in software in a very distinct context (i.e., not open-source, not a distributed processing project). The Hadoop choice allows us to build a dataset from a long-lived real-world large-scale software project. It is worth noticing that in Jira, Hadoop is reported as distinct projects - Core, HDFS, Yarn, and MapReduce - so technically, our results were based on four separate projects.

8 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 of automatic labeling of EH bugs, they comprise studies regarding EH bugs (Section 8.1) and automatic bug labeling (Section 8.2).

8.1 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 IDE.

de 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 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.

8.2 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, totaling 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 (MCC). 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 MCC 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 (KNN, MNB, SVC, and 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%.

9 Conclusion and Final Remarks

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. We also labeled 2,584 non-EH bugs analyzing their bug-fixing code, and creating a dataset composed of 7,100 bug reports.

With the dataset of EH bugs obtained, we performed a controlled experiment combining six 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 using a pre-trained DistilBERT with a linear layer trained with our proposed dataset can reasonably label EH bugs, achieving ROC-AUC scores of up to 0.88. Additionally, considering only keywords related to EH as inputs, the ML models' performance was worst compared to using all words from the textual fields when ROC-AUC

is a major concern, indicating that the full text in the textual fields it is necessary to better results.

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. It is also fundamental to evaluate in future work to which extent the practitioners benefit from using automatic labeling tools and how much their use improves the bug-fixing process. We also intend to use other approaches (e.g., semi-automatic labeling) as an avenue for future work to expand the dataset with more projects.

Data Availability The proposed dataset and all the code that supports the findings of this study are available in Figshare with the identifier <https://doi.org/10.6084/m9.figshare.22735124.v2>.

Declarations

Conflict of Interest The authors whose names are listed immediately above certify that they have NO affiliations with or involvement in any organization or entity with any financial interest (such as honoraria; educational grants; participation in speakers' bureaus; membership, employment, consultancies, stock ownership, or other equity interest; and expert testimony or patent-licensing arrangements), or non-financial interest (such as personal or professional relationships, affiliations, knowledge or beliefs) in the subject matter or materials discussed in this manuscript.

References

- Asaduzzaman M, Ahasanuzzaman M, Roy CK, Schneider KA (2016) How developers use exception handling in java? In: Proceedings of the 13th international conference on mining software repositories. ACM, New York, USA, MSR'16, pp 516–519
- Aung TWW, Wan Y, Huo H, Sui Y (2022) Multi-triage: a multi-task learning framework for bug triage. *J Syst Softw* 184:111133. <https://doi.org/10.1016/j.jss.2021.111133>, <https://www.sciencedirect.com/science/article/pii/S0164121221002302>
- Avizienis A, Laprie JC, Randell B, Landwehr C (2004) Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans Dependable Secur Comput* 1(1):11–33
- Barbosa EA, Garcia A, Barbos SDJ (2014) Categorizing faults in exception handling: a study of open source projects. In: Software engineering (SBES), 2014 Brazilian symposium on, pp 11–20
- Cacho N, Barbosa EA, Araujo J, Pranto F, Garcia A, Cesar T, Soares E, Cassio A, Filipe T, Garcia I (2014a) 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. IEEE, ICSME'14, pp 31–40
- Cacho N, César T, Filipe T, Soares E, Cassio A, Souza R, Garcia I, Barbosa EA, Garcia A (2014b) Trading robustness for maintainability: an empirical study of evolving c# programs. In: Proceedings of the 36th International Conference on Software Engineering. ICSE 2014, pp 584–595
- Catolino G, Palomba F, Zaidman A, Ferrucci F (2019) Not all bugs are the same: understanding, characterizing, and classifying bug types. *J Syst Softw* 152:165–181. <https://doi.org/10.1016/j.jss.2019.03.002>, <https://www.sciencedirect.com/science/article/pii/S0164121219300536>
- Chang BM, Choi K (2016) A review on exception analysis. *Inf Softw Technol* 77(C):1–16
- Chawla I, Singh SK (2014) Automatic bug labeling using semantic information from lsi. In: 2014 Seventh international conference on contemporary computing (IC3), pp 376–381. <https://doi.org/10.1109/IC3.2014.6897203>
- Chawla I, Singh SK (2015) An automated approach for bug categorization using fuzzy logic. In: Proceedings of the 8th India software engineering conference. Association for computing machinery, New York, NY, USA, ISEC'15, pp 90–99. <https://doi.org/10.1145/2723742.2723751>
- Chen CT, Cheng YC, Hsieh CY, Wu IL (2009) Exception handling refactorings: directed by goals and driven by bug fixing. *J Syst Softw* 82(2):333–345

- Chen H, Dou W, Jiang Y, Qin F (2019a) Understanding exception-related bugs in large-scale cloud systems. In: 2019 34th IEEE/ACM international conference on automated software engineering (ASE), pp 339–351. <https://doi.org/10.1109/ASE.2019.00040>
- Chen J, He X, Lin Q, Zhang H, Hao D, Gao F, Xu Z, Dang Y, Zhang D (2019b) Continuous incident triage for large-scale online service systems. In: Proceedings of the 34th IEEE/ACM international conference on automated software engineering. IEEE Press, ASE'19, pp 364–375. <https://doi.org/10.1109/ASE.2019.00042>
- Coelho R, Almeida L, Gousios G, Deursen AV, Treude C (2017) Exception handling bug hazards in android. *Empirical Softw Engg* 22(3):1264–1304
- Cohen J (1960) A coefficient of agreement for nominal scales. *Educ Psychol Measur* 20:37–46
- Dalton F, Ribeiro M, Pinto G, Fernandes L, Gheyri R, Fonseca B (2020) Is exceptional behavior testing an exception? an empirical assessment using java automated tests. In: Proceedings of the evaluation and assessment in software engineering. Association for computing machinery, New York, NY, USA, EASE'20, pp 170–179. <https://doi.org/10.1145/3383219.3383237>
- de Pádua GB, Shang W (2017) Studying the prevalence of exception handling anti-patterns. In: Proceedings of the 25th international conference on program comprehension. IEEE Press, Piscataway, NJ, USA, ICPC'17, pp 328–331
- de Pádua GB, Shang W (2018) Studying the relationship between exception handling practices and post-release defects. In: Proceedings of the 15th international conference on mining software repositories. ACM, New York, NY, USA, MSR'18, pp 564–575. <https://doi.org/10.1145/3196398.3196435>
- de Sousa DBC, Maia PHM, Rocha LS, Viana W (2020) Studying the evolution of exception handling anti-patterns in a long-lived large-scale project. *J Braz Comput Soc* 26(1):1. <https://doi.org/10.1186/s13173-019-0095-5>
- Ebert F, Castor F, Serebrenik A (2015) An exploratory study on exception handling bugs in java programs. *J Syst Softw* 106(C):82–101
- Ebert F, Castor F, Serebrenik A (2020) 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), pp 552–556. <https://doi.org/10.1109/SANER48275.2020.9054791>
- Elzanaty F, Rezk C, Lijbrink S, van Bergen W, Cote M, McIntosh S (2021) Automatic recovery of missing issue type labels. *IEEE Softw* 38(3):35–42. <https://doi.org/10.1109/MS.2020.3004060>
- Filho JLM, Rocha L, Andrade R, Britto R (2017) Preventing erosion in exception handling design using static-architecture conformance checking. In: Proceedings of the 11th European conference on software architecture. Springer International Publishing, Cham, ECSA'17, pp 67–83. https://doi.org/10.1007/978-3-319-65831-5_5
- Gallardo R, Hommel S, Kannan S, Gordon J, Zakhour SB (2014) The Java tutorial: a short course on the basics, 6th edn. Addison-Wesley Professional, Java Series
- Garcia AF, Rubira CM, Romanovsky A, Xu J (2001) A comparative study of exception handling mechanisms for building dependable object-oriented software. *J Syst Softw* 59(2):197–222
- Goffi A, Gorla A, Ernst MD, Pezzè M (2016) Automatic generation of oracles for exceptional behaviors. In: Proceedings of the 25th international symposium on software testing and analysis. ACM, New York, NY, USA, ISSTA 2016, pp 213–224
- Gomes LAF, da Silva Torres R, Côrtes ML (2019) Bug report severity level prediction in open source software: a survey and research opportunities. *Inf Softw Technol* 115:58–78. <https://doi.org/10.1016/j.infsof.2019.07.009>, <https://www.sciencedirect.com/science/article/pii/S0950584919301648>
- Goodenough JB (1975) Exception handling: issues and a proposed notation. *Commun ACM* 18:683–696
- Hu H, Zhang H, Xuan J, Sun W (2014) Effective bug triage based on historical bug-fix information. In: Proceedings of the 2014 IEEE 25th international symposium on software reliability engineering. IEEE Computer Society, USA, ISSRE'14, pp 122–132. <https://doi.org/10.1109/ISSRE.2014.17>
- Kechagia M, Spinellis D (2014) Undocumented and unchecked: exceptions that spell trouble. In: Proceedings of the 11th working conference on mining software repositories. ACM, New York, USA, MSR 2014, pp 312–315
- Kienzle J (2008) On exceptions and the software development life cycle. In: Proceedings of the 4th international workshop on exception handling. ACM Press, New York, NY, USA, WEH'08, pp 32–38
- Knudsen J (1987) Better exception-handling in block-structured systems. *IEEE Softw* 4(3):40–49. <https://doi.org/10.1109/MS.1987.230705>
- Köksal O, Öztürk CE (2022) A survey on machine learning-based automated software bug report classification. In: 2022 International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT), pp 635–640. <https://doi.org/10.1109/ISMSIT56059.2022.9932822>
- Lee SR, Heo MJ, Lee CG, Kim M, Jeong G (2017) Applying deep learning based automatic bug triager to industrial projects. In: Proceedings of the 2017 11th joint meeting on foundations of software engineering.

- Association for computing machinery, New York, NY, USA, ESEC/FSE 2017, pp 926–931. <https://doi.org/10.1145/3106237.3117776>
- Lima LP, Rocha LS, Bezerra CIM, Paixao M (2021) Assessing exception handling testing practices in open-source libraries. *Empir Softw Eng* 26(5):85. <https://doi.org/10.1007/s10664-021-09983-3>
- Marcilio D, Furia CA (2021) How java programmers test exceptional behavior. In: 2021 IEEE/ACM 18th International conference on mining software repositories (MSR), pp 207–218. <https://doi.org/10.1109/MSR52588.2021.00033>
- Marinescu C (2011) Are the classes that use exceptions defect prone? In: Proceedings of the 12th international workshop on principles of software evolution and the 7th annual ERCIM workshop on software evolution. ACM, pp 56–60
- Marinescu C (2013) Should we beware the exceptions? an empirical study on the eclipse project. In: Symbolic and numeric algorithms for scientific computing (SYNASC), 2013 15th international symposium on, IEEE, pp 250–257
- Melo H, Coelho R, Treude C (2019) Unveiling exception handling guidelines adopted by java developers. In: 2019 IEEE 26th International conference on software analysis, evolution and reengineering (SANER), pp 128–139. <https://doi.org/10.1109/SANER.2019.8668001>
- Miller R, Tripathi A (1997) Issues with exception handling in object-oriented systems. In: Aksit M, Matsuoka S (eds) ECOOP'97 - Object-Oriented Programming. Lecture Notes in Computer Science, vol 1241. Springer, Berlin / Heidelberg, pp 85–103
- Pandey N, Sanyal DK, Hudait A, Sen A (2017) Automated classification of software issue reports using machine learning techniques: an empirical study. *Innov Syst Softw Eng* 13(4):279–297. <https://doi.org/10.1007/s11334-017-0294-1>
- Parnas DL, Würges H (1976) Response to undesired events in software systems. In: Proceedings of the 2nd international conference on software engineering. IEEE Computer Society Press, Los Alamitos, CA, USA, ICSE'76, pp 437–446
- Peters F, Tun TT, Yu Y, Nuseibeh B (2019) Text filtering and ranking for security bug report prediction. *IEEE Trans Software Eng* 45(6):615–631. <https://doi.org/10.1109/TSE.2017.2787653>
- Picus O, Serban C (2022) Bugsby: a tool support for bug triage automation. In: Proceedings of the 2nd ACM international workshop on AI and software testing/analysis. Association for computing machinery, New York, NY, USA, AISTA 2022, pp 17–20. <https://doi.org/10.1145/3536168.3543301>
- Robillard MP, Murphy GC (2003) Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Trans Softw Eng Methodol* 12(2):191–221. <https://doi.org/10.1145/941566.941569>
- Sanh V, Debut L, Chaumond J, Wolf T (2019) Distilbert, a distilled version of BERT: smaller, faster, cheaper and lighter. [arXiv:1910.01108](https://arxiv.org/abs/1910.01108)
- Sawadpong P, Allen EB (2016) Software defect prediction using exception handling call graphs: a case study. In: High assurance systems engineering (HASE), 2016 IEEE 17th international symposium on, IEEE, pp 55–62
- Sawadpong P, Allen EB, Williams BJ (2012) Exception handling defects: an empirical study. In: High-assurance systems engineering (HASE), 2012 IEEE 14th International symposium on, IEEE, pp 90–97
- Shah H, Gorg C, Harrold MJ (2010) Understanding exception handling: viewpoints of novices and experts. *IEEE Trans Softw Eng* 36(2):150–161
- Shahroki A, Feldt R (2013) A systematic review of software robustness. *Inf Softw Technol* 55(1):1–17
- Sinha S, Harrold MJ (2000) Analysis and testing of programs with exception handling constructs. *IEEE Trans Software Eng* 26(9):849–871. <https://doi.org/10.1109/32.877846>
- Spadini D, Aniche M, Bacchelli A (2018) 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. ACM Press, New York, USA, pp 908–911. <https://doi.org/10.1145/3236024.3264598>, <http://dl.acm.org/citation.cfm?doid=3236024.3264598>
- Sparck Jones K (1988) A statistical interpretation of term specificity and its application in retrieval. Taylor Graham Publishing, GBR, pp 132–142
- Tian Y, Lo D, Xia X, Sun C (2015) Automated prediction of bug report priority using multi-factor analysis. *Empir Softw Eng* 20(5):1354–1383. <https://doi.org/10.1007/s10664-014-9331-y>
- Uddin J, Ghazali R, Deris MM, Naseem R, Shah H (2017) A survey on bug prioritization. *Artif Intell Rev* 47(2):145–180. <https://doi.org/10.1007/s10462-016-9478-6>
- Vieira RG, Mattos CLC, Rocha LS, Gomes JPP, Paixão M (2022) The role of bug report evolution in reliable fixing estimation. *Empir Softw Eng* 27(7):164. <https://doi.org/10.1007/s10664-022-10213-7>
- Vieira R, da Silva A, Rocha L, Gomes JaP (2019) 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. Association for Computing Machinery, New York, USA, PROMISE'19, pp 80–89. <https://doi.org/10.1145/3345629.3345639>
- White T (2015) Hadoop: the definitive guide: storage and analysis at internet scale, 4th edn. O'Reilly Media
- Wohlin C, Runeson P, Hst M, Ohlsson MC, Regnell B, Wessln A (2012) Experimentation in software engineering. Springer Publishing Company, Incorporated
- Xu B, Huang JZ, Williams G, Wang Q, Ye Y (2012) Classifying very highdimensional data with random forests built from small subspaces. *Int J Data Warehous Min* 8(2):44–63
- Zhang P, Elbaum S (2014) Amplifying tests to validate exception handling code: an extended study in the mobile application domain. *ACM Trans Softw Eng Methodol* 23(4):32:1-32:28
- Zhang J, Wang X, Zhang H, Sun H, Pu Y, Liu X (2021) Learning to handle exceptions. In: Proceedings of the 35th IEEE/ACM international conference on automated software engineering. Association for Computing Machinery, New York, USA, ASE'20, pp 29–41. <https://doi.org/10.1145/3324884.3416568>

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.

Authors and Affiliations

Antônio José A. da Silva¹ · Renan G. Vieira¹  · Diego P. P. Mesquita² · João Paulo P. Gomes¹ · Lincoln S. Rocha¹

✉ Renan G. Vieira
renan@dc.ufc.br

Antônio José A. da Silva
amanciosilva@great.ufc.br

Diego P. P. Mesquita
diego.mesquita@fgv.br

João Paulo P. Gomes
jpaulo@dc.ufc.br

Lincoln S. Rocha
lincoln@dc.ufc.br

¹ Federal University of Ceará, Av. Humberto Monte, s/n - Pici, Fortaleza - CE 60440-593, Brazil

² Getulio Vargas Foundation, Praia do Botafogo, 190 - Botafogo, Rio de Janeiro - RJ 22250-900, Brazil