



Titre: An Empirical Study of the Causes and Symptoms of Machine Learning Exceptions
Title:

Auteur: Amin Ghadesi
Author:

Date: 2023

Type: Mémoire ou thèse / Dissertation or Thesis

Référence: Ghadesi, A. (2023). An Empirical Study of the Causes and Symptoms of Machine Learning Exceptions [Mémoire de maîtrise, Polytechnique Montréal]. PolyPublie.
Citation: <https://publications.polymtl.ca/57116/>

 **Document en libre accès dans PolyPublie**
Open Access document in PolyPublie

URL de PolyPublie: <https://publications.polymtl.ca/57116/>
PolyPublie URL:

Directeurs de recherche: Heng Li, & Maxime Lamothe
Advisors:

Programme: Génie informatique
Program:

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

**An Empirical Study of the Causes and Symptoms of Machine Learning
Exceptions**

AMIN GHADESI

Département de génie informatique et génie logiciel

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

Décembre 2023

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

Ce mémoire intitulé :

**An Empirical Study of the Causes and Symptoms of Machine Learning
Exceptions**

présenté par **Amin GHADESI**

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

Zohreh SHARAFI, présidente

Heng LI, membre et directeur de recherche

Maxime LAMOTHE, membre et codirecteur de recherche

Bram ADAMS, membre

DEDICATION

*I dedicate this to my mother and aunts
for your constant love and support . . .*

ACKNOWLEDGEMENTS

I extend my heartfelt appreciation to my research director, Professor Heng Li, for their exceptional guidance, unwavering support, and insightful supervision throughout the research process. Their expertise has been instrumental in shaping the direction and quality of this thesis.

I am equally grateful to my co-director, Professor Maxime Lamothe, whose valuable contributions and collaborative spirit greatly enriched this research. Their feedback and encouragement were indispensable, and I am thankful for the mentorship they provided.

A special acknowledgment goes to the members of the MOOSE and SWAT labs. Working alongside such a dedicated and talented group of individuals has been an enriching experience. The collaborative atmosphere and shared commitment to excellence have significantly contributed to the success of this project.

I would like to express my gratitude to each committee member— Professor Bram Adams and Professor Zohreh Sharafi for accepting to evaluate my master’s thesis.

RÉSUMÉ

L'apprentissage machine (ML), y compris l'apprentissage profond, a récemment gagné en popularité dans un large éventail d'applications. Toutefois, comme les logiciels traditionnels, les applications d'apprentissage automatique ne sont pas à l'abri des bogues résultant d'erreurs de programmation. Les défaillances programmatiques se manifestent généralement par des exceptions au cours de l'exécution du programme. Comprendre les causes et les symptômes des exceptions de ML peut offrir aux développeurs des informations précieuses pour améliorer leurs pratiques de programmation et prévenir de telles exceptions.

Cette thèse vise à ouvrir la voie à de futures recherches axées sur le développement de solutions pour aider les développeurs de ML à prévenir ou à réduire les exceptions. Malgré des études antérieures portant sur les exceptions dans les logiciels traditionnels, il n'y a pas d'exploration systématique des exceptions en ML dans la littérature existante. Par conséquent, cette thèse mène une étude empirique complète des questions de Stack Overflow (SO) liées aux exceptions de ML, cherchant à comprendre les causes et les symptômes associés à ces exceptions.

La première partie de cette thèse explore les causes des exceptions de ML en analysant les traces de pile des applications de ML postées sur SO. Ces traces de pile décrivent la chaîne d'appels de fonction qui conduit à une situation anormale ou à une exception. En effet, ces exceptions peuvent traverser toute la pile logicielle (y compris les applications et les bibliothèques). Ainsi, l'étude des modèles liés au ML dans les traces de pile peut aider les praticiens et les chercheurs à comprendre les causes des exceptions dans les applications de ML et les défis auxquels sont confrontés les développeurs de ML. A cette fin, nous exploitons SO et étudions 11,449 les traces de piles liées à la ML de sept bibliothèques ML Python populaires. Tout d'abord, nous observons que les questions de ML qui contiennent des traces de pile sont moins susceptibles d'obtenir des réponses acceptées que les questions qui n'en contiennent pas, même si elles reçoivent plus d'attention (c.-à-d. plus de vues et de commentaires).

Deuxièmement, nous observons que des motifs récurrents existent dans les traces de pile de ML, même à travers différentes bibliothèques de ML, avec une petite partie des motifs couvrant de nombreuses traces de pile. Troisièmement, nous déduisons cinq catégories de haut niveau et 25 types de bas niveau des traces de pile : la plupart des modèles sont liés à *syntaxe de base de Python*, *formation de modèle*, *parallélisation*, *transformation de données*, et *invocation d'un sous-processus*. En outre, les motifs liés à *invocation de sous-processus*, *exécution de module externe* et *appel d'API à distance* sont parmi les moins susceptibles

d’obtenir des réponses acceptées sur SO.

Dans la deuxième partie de la thèse, nous étudions les symptômes rapportés des exceptions et explorons leur influence sur le soutien de la communauté dans la résolution de ces exceptions. Comme pour la première partie, nous avons mené cette investigation sur sept bibliothèques de ML sur SO. Cependant, contrairement à la première partie, nous avons étendu notre attention au-delà des piles d’appels pour inclure d’autres aspects des exceptions, en particulier le type d’exception et le message d’erreur. Plus précisément, la deuxième étude montre la distribution des symptômes d’exception ML sur SO, constatant qu’environ la moitié des questions contiennent les trois symptômes tandis que d’autres présentent des combinaisons spécifiques. L’étude de l’ambiguïté des symptômes révèle que les modèles de messages d’erreur individuels ou les types d’exception n’identifient pas sans équivoque les questions liées à l’exception ML. L’analyse indique que l’affichage de modèles de messages d’erreur individuels ou de types d’exception ne permet pas d’identifier distinctement les questions liées aux exceptions ML. Il est intéressant de noter que les questions ne présentant que des symptômes de pile d’appel sont moins susceptibles de recevoir un soutien rapide de la part de la communauté. Cependant, l’ajout de types d’exception et de messages d’erreur aux piles d’appels augmente la probabilité et la rapidité d’obtention de réponses acceptées.

Nos résultats fournissent des indications aux chercheurs, aux développeurs de bibliothèques de ML et aux modérateurs de forums techniques afin de mieux aider les développeurs de ML à écrire du code ML sans erreur. Par exemple, les recherches futures pourraient exploiter les schémas communs des traces de piles pour aider les développeurs de ML à trouver des solutions à des problèmes similaires aux leurs ou à identifier des experts qui ont l’expérience de la résolution de schémas de problèmes similaires. Les chercheurs et les développeurs de bibliothèques de ML pourraient s’efforcer en priorité d’aider les développeurs de ML à identifier les mauvaises utilisations des API de ML, les incohérences dans les formats de données et les contestations potentielles de données/ressources, afin que les développeurs de ML puissent mieux éviter/corriger les schémas d’exception liés au modèle, les schémas d’exception liés aux données et les schémas d’exception liés au multiprocessus, respectivement. Notre étude sur les symptômes des exceptions de ML implique que les développeurs devraient publier des informations non ambiguës pour identifier les exceptions de ML et recevoir le soutien de la communauté.

ABSTRACT

Machine learning (ML), including deep learning, has recently gained tremendous popularity in a wide range of applications. However, like traditional software, ML applications are not immune to the bugs that result from programming errors. Programmatic failures typically manifest as exceptions during program execution. Understanding the root causes and symptoms of ML exceptions can offer developers valuable insights to enhance their programming practices and prevent such exceptions.

This thesis aims to pave the way for future investigations focused on developing solutions to aid ML developers in preventing or diminishing exceptions. Despite previous studies delving into exceptions in traditional software, there is a lack of systematic exploration of ML exceptions in the existing literature. Consequently, this thesis conducts a comprehensive empirical study of Stack Overflow (SO) questions related to ML exceptions, seeking to comprehend the causes and symptoms associated with these exceptions.

The first part of this thesis explores the causes of ML exceptions by analyzing the stack traces of ML applications posted on SO. These stack traces describe the chain of function calls that lead to an anomalous situation or exception. Indeed, these exceptions may cross the entire software stack (including applications and libraries). Thus, studying the ML-related patterns in stack traces can help practitioners and researchers understand the causes of exceptions in ML applications and the challenges faced by ML developers. To that end, we mine SO and study 11,449 ML-related stack traces related to seven popular Python ML libraries. First, we observe that ML questions that contain stack traces are less likely to get accepted answers than questions that don't, even though they gain more attention (i.e., more views and comments). Second, we observe that recurrent patterns exist in ML stack traces, even across different ML libraries, with a small portion of patterns covering many stack traces. Third, we derive five high-level categories and 25 low-level types from the stack trace patterns: most patterns are related to *python basic syntax*, *model training*, *parallelization*, *data transformation*, and *subprocess invocation*. Furthermore, the patterns related to *subprocess invocation*, *external module execution*, and *remote API call* are among the least likely to get accepted answers on SO.

In the second part of the thesis, we study the reported symptoms of exceptions and explore their influence on community support in resolving these exceptions. Similar to the initial part of the inquiry, we conducted this investigation across seven ML libraries on SO. However, in contrast to the first part, we extended our focus beyond call stacks to include other aspects of

exceptions, specifically, the exception type and error message. Specifically, the second study shows the distribution of ML exception symptoms on SO, finding that around half of the questions contain all three symptoms while others exhibit specific combinations. The investigation into symptom ambiguity reveals that individual error message templates or exception types do not unequivocally identify ML exception-related questions. The analysis indicates that posting individual error message templates or exception types does not distinctly identify ML exception-related questions. Interestingly, questions with only call stack symptoms are less likely to receive prompt community support. However, adding exception types and error messages to call stacks increases the likelihood and speed of obtaining accepted answers.

Our findings provide insights for researchers, ML library developers, and technical forum moderators to better support ML developers to write error-free ML code. For example, future research can leverage the common patterns of stack traces to help ML developers locate solutions to problems similar to theirs or to identify experts who have experience solving similar patterns of problems. Researchers and ML library developers could prioritize efforts to help ML developers identify misuses of ML APIs, mismatches in data formats, and potential data/resource contentions, so that ML developers can better avoid/fix model-related exception patterns, data-related exception patterns, and multi-process related exception patterns, respectively. Our study on the symptoms of ML exceptions implies that developers should post unambiguous information to identify ML exceptions and receive community support.

The second study demonstrates the symptoms of exceptions and assesses their impact on community support in resolving failures. Our focus extends beyond call stacks to contain additional aspects of exceptions, specifically the exception type and error message. This study shows the distribution of ML exception symptoms on Stack Overflow, highlighting that approximately half of the questions involve all three symptoms, while others feature specific combinations. The exploration of symptom ambiguity indicates that individual error message templates or exception types do not definitively identify ML exception-related questions. Notably, questions solely featuring call stack symptoms are less likely to receive prompt community support. However, incorporating exception types and error messages alongside call stacks enhances the likelihood and speed of obtaining accepted answers. We hope our results can help improve how exceptions are used on forums such as Stack Overflow.

TABLE OF CONTENTS

DEDICATION	iii
ACKNOWLEDGEMENTS	iv
RÉSUMÉ	v
ABSTRACT	vii
TABLE OF CONTENTS	ix
LIST OF TABLES	xi
LIST OF FIGURES	xii
LIST OF SYMBOLS AND ACRONYMS	xiii
CHAPTER 1 INTRODUCTION	1
1.1 Context and Motivation	1
1.2 Research Questions	2
1.3 Contributions	3
1.4 Thesis Plan	4
CHAPTER 2 LITERATURE REVIEW	5
CHAPTER 3 METHODOLOGY	9
3.1 Overview	9
3.2 Subject Data and Data Collection	9
CHAPTER 4 STUDYING THE CAUSES OF ML EXCEPTIONS	10
4.1 Introduction	10
4.2 Overview	13
4.3 Subject Data	13
4.4 Data Collection	14
4.5 Data Analysis	15
4.6 Research Questions and Results	17
4.7 Threats to Validity	37

CHAPTER 5	STUDYING THE SYMPTOMS OF ML EXCEPTIONS	44
5.1	Introduction	44
5.2	Overview	46
5.3	Subject Data	46
5.4	Data Collection	47
5.5	Data Analysis	49
5.6	Research Questions and Results	52
5.7	Threats to Validity	68
CHAPTER 6	CONCLUSION	70
REFERENCES		72

LIST OF TABLES

Table 4.1	The number of questions on SO for each studied ML library, including all questions, those that contain <i>Code Blocks</i> and those that contain stack traces	19
Table 4.2	The distribution of stack traces on SO for our studied ML libraries .	19
Table 4.3	The median question and code lengths for the SO posts related to our studied ML libraries	20
Table 4.4	The median community support variables for SO posts of our studied ML libraries	39
Table 4.5	The median time to answer for SO posts of our studied ML libraries .	40
Table 4.6	What percent of questions and stack traces fall into the pattern set ($support \geq 2$)	41
Table 4.7	The number of shared stack trace patterns amongst various ML libraries ($support \geq 2$)	41
Table 4.8	Classification of 995 stack trace patterns	42
Table 4.9	Statistical information of pattern types	43
Table 5.1	The volume of the studied ML-related libraries	48
Table 5.2	ML library tags	49
Table 5.3	ML exception symptoms abbreviations and meanings	51
Table 5.4	The distribution of symptom combinations	55
Table 5.5	Top 20 most frequent Exception Types	57
Table 5.6	Distribution of Exception Types per Error Message Template and vice versa	61
Table 5.7	Distribution of Call Stacks for each Exception Type and Error Message Template	62
Table 5.8	Top 30 most frequent Error Message Templates	63
Table 5.9	The percentage of questions receiving an accepted answer for different exception symptom combinations.	66

LIST OF FIGURES

Figure 3.1	A general overview of our studies	9
Figure 4.1	An overview of our study design	13
Figure 4.2	A truncated example of a SO question. (ID=52253378)	15
Figure 4.3	A truncated example of stack trace transformation. (ID=70716638) .	16
Figure 4.4	The cumulative percentage of stack trace patterns for various supports (<i>support</i> ≥ 1)	22
Figure 4.5	The cumulative percentage of stack trace patterns vs. pattern's length (<i>support</i> ≥ 1).	23
Figure 4.6	The percentage of covered questions based on stack trace patterns (pat- terns are sorted based on their supports).	24
Figure 4.7	Stack trace pattern correlation between ML-libraries. The heatmap plot displays the extent of shared patterns, ranging from 0 (light green) to 2044 (dark green), based on the intensity of color.	25
Figure 5.1	An overview of the study design	47
Figure 5.2	A truncated example of a SO post. (ID=60798712)	50
Figure 5.3	A truncated example of call stack transformation. (ID=70716638) . .	52
Figure 5.4	The trend of questions with at least one Error Message and without any Call Stack	54
Figure 5.5	The cumulative percentage of ML-related questions vs. the percentage of Exception types	56
Figure 5.6	The cumulative percentage of ML-related questions vs. the percentage of Error Message Templates	58
Figure 5.7	The level of community support for questions with different combina- tions of exception symptoms.	66
Figure 5.8	The status of community support among questions with and without similarity between Error Message and the tile of exception questions. Numbers on the box plots illustrate their medians.	67

LIST OF SYMBOLS AND ACRONYMS

RQ	Research Question
Q&A	Question and Answer
SO	Stack Overflow
ML	Machine Learning
CC-Span	Closed Contiguous Sequential Patterns
CS	Call Stack
ExT	Exception Type
EM	Error Message
EMT	Error Message Template
LOC	Line Of Code
IQR	Interquartile Range

CHAPTER 1 INTRODUCTION

1.1 Context and Motivation

The popularity of machine learning (ML) (including deep learning) applications has grown rapidly in recent years [1, 2]. Indeed, a 2020 Deloitte survey of 2,737 IT companies from nine countries found that modern machine learning technologies were being used by more than 50% of these companies, and that 95% of the respondents were planning to use them within the following year [3]. Furthermore, in 2022, The NewVantage Partners’ *Data and AI Leadership Executive Survey* demonstrated that 91.7% of organizations were expanding their investment in data and AI activities, at the core of which is ML [2].

The growing trend in ML adoption delivers new challenges for ML developers (i.e., software developers and data scientists) [4–7]. When ML developers face issues in their development process, they can turn to technical question and answer (Q&A) forums such as Stack Overflow (SO) to answer their questions [5–9]. In this work, we analyze such questions to understand the failures (more specifically, exceptions) that technical forum moderators face when they develop ML programs. We target ML-related questions on SO because, with over 22 million questions, 33 million answers, and roughly 18 million users, SO is the largest technical Q&A forum [10]. A 2021 survey on SO [11] has shown that 80 percent of respondents visit the forum weekly, and 50 percent visit it daily. We, therefore, leverage SO to obtain data on exceptions that developers face when they develop ML programs because it is both popular and ingrained in developer work habits.

Prior works have studied the issues and challenges of developing machine learning applications [4–8, 12]. Indeed, it has been shown that the root causes of bugs and ML library challenges can be uncovered through a combination of bug reports and ML posts from GitHub and SO. However, the exceptions of ML applications remain unstudied.

Studying the causes of ML exceptions. A stack trace reports the chain of function calls at play when an anomalous situation, or exception, occurs. At the time of the exception, these function calls are either under execution or waiting for other function calls to be completed. The stack traces provide clues for understanding the errors that lead to exceptions (i.e., what chain of function calls lead to the exception). In this work, we study the stack traces of ML applications to understand the causes of exceptions in ML applications. We thus aim to provide insights to engage ML community support and leverage existing ML-related stack traces to improve current ML development practices. Our work provides a comple-

mentary perspective to existing works that study the issues and challenges of developing ML applications.

In the study of ML exception causes we study ~11K ML-related stack traces posted in SO questions, aiming to understand common patterns that cause exceptions in ML applications. To identify ML-related SO questions, we consider questions related to the use of seven popular Python ML libraries, including TensorFlow [13], Keras [14], Scikit-learn [15], PyTorch [16], NLTK [17], HuggingFace [18], and Spark ML [19]. We focus on the Python language as it is the dominant programming language for ML application development [20, 21]. We perform quantitative and qualitative analyses to understand the characteristics of SO questions with stack traces and to analyze the patterns of stack traces therein. Through our analyses of ML-related stack traces, we aim to provide insights for researchers, ML library developers, and technical forum moderators to help ML developers improve current ML development practices.

Studying the symptoms of ML exceptions. We analyze the symptoms of ML exceptions and explore their impact on community support. Similar to the first part of the study, we conducted this investigation across seven ML libraries on SO. However, in contrast to the first part, we extended our focus beyond call stacks to include other aspects of exceptions, specifically, the exception type and error message.

1.2 Research Questions

In the study of ML exception causes, we structure our study and findings along the following three research questions (RQs):

- **RQ1. What are the characteristics of ML-related posts with stack traces?** To understand the significance of stack traces and their impact, we study the prevalence of SO questions with stack traces and how they impact community support on SO. We observe that 12.6% of posts related to our studied libraries include a stack trace. Although these posts receive more attention, they are less likely to have accepted answers, and take longer to be answered.
- **RQ2. What are the characteristics of the stack trace patterns in ML-related questions?** To understand the causes of ML application exceptions, we mine the common patterns in the ML stack traces posted on SO. We observe that most patterns are short (80% of the patterns contain fewer than five calls), and 19.53% are shared among at least two ML libraries. We also find that a small percentage of patterns (20%) span a large number of questions (up to 85%).

- **RQ3. What are the categories of the stack trace patterns, and which categories are most challenging for developers?** Through a qualitative study, we identify five high-level categories and 25 low-level types of stack trace patterns associated with ML development. We find that exceptions are often caused by misunderstandings or uncertainties related to ML API usage, data formats, or language constructs. Furthermore, exceptions related to external dependencies or manipulations of artifacts are less likely to receive timely community support.

On the other hand, in the second study, which is related to ML exception symptoms, our RQs are:

- **RQ1. What are the distributions of ML exception symptoms posted on Stack Overflow?** About half of the exception-related ML questions (i.e., containing at least one exception symptom) contain all three exception symptoms (call stacks, exception types, and error messages). Most of the other exception-related ML questions either contain a call stack or a combination of an exception type and an error message.
- **RQ2. How ambiguous are the different symptoms of ML exceptions?** Posting error message templates or exception types individually cannot unambiguously identify exception-related questions. Besides, an exception type or an error message template can be associated with completely different call stacks. Thus, we suggest developers post the complete symptoms of ML exceptions to reduce the ambiguity of their posts. Post searchers should also leverage as many symptoms as possible to maximize their chance of uniquely identifying similar questions related to ML exceptions.
- **RQ3. How do the exception symptoms impact community support?** Surprisingly, ML questions with all three exception symptoms are not the most likely to receive timely community support. In fact, questions with only the call stack symptoms are least likely to receive a timely answer, and adding exception types and error messages to the call stacks can increase the likelihood and speed of receiving an accepted answer.

1.3 Contributions

The main contributions of our studies include:

- The first study is the first step towards understanding the stack traces of machine learning applications and their related issues posted on SO.

- We observe that questions with stack traces are less likely to get timely accepted answers. Technical forum (e.g., SO) moderators and researchers should pay particular attention to these questions to help developers.
- We observe patterns in stack traces of ML applications, and find that a small subset of them cover a wide range of questions, even across different libraries. Focusing on these prevalent patterns could produce impactful improvements.
- We provide a hierarchical taxonomy of the stack trace patterns of ML applications and recommend actions to improve ML applications and their usage.
- Our observations identify the most challenging pattern types and provide critical insight for future research.
- The second study marks a first step in understanding the exceptions of machine learning applications and the associated issues posted on Stack Overflow.
- Through an empirical evaluation of Stack Overflow posts, we provide actionable recommendations for posters and forum moderators to improve current practices, and maximize community support.
- We provide a database containing machine learning exceptions sourced from Stack Overflow.

1.4 Thesis Plan

The rest of the thesis is organized as follows: Chapter 2 surveys prior works related to our studies. Chapter 3 describes the methodology of the thesis. Chapter 4 explains the first study about the cause of ML exceptions, followed by Chapter 5, which clarifies our second study concerning the symptoms of ML exceptions. Finally, Chapter 6 concludes the thesis.

CHAPTER 2 LITERATURE REVIEW

In this chapter, we discuss prior works related to our thesis, including studies on software stack traces, ML-related forum posts, failures in ML applications, and exception handling.

Studies on ML-related forum posts. With a significant increase in the usage of machine learning systems in recent years, a lot of research has been done on the Q&A forums, such as SO, to understand the state and challenges of ML [5–9]. Alshangiti et al. [6] and Hamidi et al. [7] perform empirical studies on SO posts related to ML to find developers’ challenges in developing ML models. They find that most of the questions are related to model deployment phases and that there is a lack of ML experts in the SO community.

Zhang et al. [5] focus on Deep Learning (DL) applications and study the existing challenges in developing DL projects by analyzing the commonly asked questions and their answers on SO. Besides, Bangash et al. [8] and Islam et al. [9] select posts related to ML to investigate developers’ understanding of ML libraries. The latter focuses on 10 popular ML libraries such as Tensorflow and Keras and suggests urgently needed research in this area. In another study, Islam et al. [4] analyzed around 2k posts on SO and 500 bug fix commits from the GitHub platform with regards to the 5 DL libraries, including Caffe, Keras, Tensorflow, Theano, and Torch, as well as categorize the characteristics of bugs. Similar to the existing studies, our study focuses on SO posts with tags related to 7 ML libraries. However, different from these studies, we focus on the error patterns of ML applications manifested in stack traces. Alaleh et al. [22] analyzed 43,950 Stack Overflow posts from 2008 to 2020, examining the evolution of ML-related questions across six programming languages, investigating changes in the number of posts over time, their distribution across different phases of an ML pipeline, and the varying levels of difficulty associated with posts from different languages or pipeline phases.

Islam et al. [23] examined 3,243 highly-rated Q&A posts on Stack Overflow related to ten ML libraries (including *TensorFlow*, *Keras*, *Scikit-learn*, *Weka*, *Caffe*, *Theano*, *MMLib*, *Torch*, *Mahout*, and *H2O*). The questions were classified into seven stages (corresponding to a typical ML pipeline [24]) to analyze the correlation between libraries and stages. The study then proceeded to identify the most challenging stage, comprehend the nature of problems, scrutinize library characteristics, and assess the consistency of difficulties over time. Samantha et al. [25] discuss the concept of contracts in the context of ML APIs for four popular ML libraries, including *TensorFlow*, *Scikit-learn*, *Keras*, and *PyTorch* as well as posts from SO. Such contracts specify the expected behavior of an API and allow users to use the API

correctly in their codes. They present several insights, including the challenges ML API users face. They highlight issues with temporal method orders, input-related contract violations, inadequate error messages, and a high percentage of contract violations in early ML pipeline stages. Henghui et al. [26] conduct a comprehensive study of the issues deep learning developers face by studying the SO dataset. They uncovered 30 distinct topics and assessed their complexity and popularity, highlighting Gradient Propagation as the most popular and Object Detection as the most challenging. The analysis also determines typical problems across five deep learning frameworks, TensorFlow, Torch, Caffe, Theano, and Keras, and examines topic trends. In contrast to these studies, our research examines forum posts about ML exceptions.

Studies on ML library usages. Numerous ML libraries are available for developers to build their ML systems. Understanding the use case of each of these libraries helps developers choose the best fit for their ecosystem. Dilhara et al. [27] conduct an empirical study on the usage of ML libraries by ML developers. They notice (1) a growing tendency to use ML libraries in 2018 compared to 2013, and (2) usage of multiple ML libraries in the implementation of “ML workflows”. Majidi et al. [28] realize that multiple automated machine learning libraries are infrequently used in scripts and projects. Also, Humbatova et al. [29] provide a taxonomy of the existing faults in applications that are based on DL. In addition, Zhang et al. [30] present an empirical study on the DL program failures by focusing on TensorFlow, PyTorch, MXNet, and four related toolkit libraries, including NumPy, DLTK, Detectron, and Fairseq. The input data comes from a DL application in Microsoft, and they manually categorize 400 failure messages to identify the common root causes of failures. Our work complements the existing studies by studying the SO posts; specifically, seven widespread Python ML libraries, including TensorFlow, Keras, Scikit-learn, PyTorch, NLTK, Hugging Face, and Spark ML, to find patterns in stack traces and their challenges.

Studies on bugs in ML applications. In software engineering, failure is commonly defined as the inability of a system or its components to meet specified functions [31]. Similarly, in ML-based systems, faults can result in poor performance, data corruption, crashes, and memory out-of-bounds, all recognized as failures in such systems [31, 32]. An ML failure can occur in the libraries, programmer code, or data. So, researchers study the failures in each area separately [32, 33]. Based on the types of failures, there are three different testing levels for ML-based systems: model testing, integration testing, and system testing. In addition, several studies focus on creating bug benchmarks in ML-based systems [32, 34–36]. For instance, Morovati et al. [32] explored the reproducibility and verifiability of bugs in ML-based systems on GitHub and SO. They identified crucial factors affecting each aspect

and overcame challenges by establishing a benchmark named *defect4ML*, containing 100 bugs reported by ML developers in *TensorFlow* and *Keras*. The goal was to establish a standard benchmark for evaluating the effectiveness of tools and techniques in ML-based systems. The black box and stochastic nature of ML applications and frameworks [37] pose a challenge for developers, making it difficult to comprehend how the models operate and make decisions.

Florian et al. [38] focused on a subset of bugs known as silent bugs. These bugs represent incorrect behavior without causing system crashes or displaying error messages. The researchers conducted the first empirical study on silent bugs in *TensorFlow*'s high-level API *Keras* from the GitHub repository. They determined 77 (out of 1,168 issues) reproducible silent bugs affecting programs, categorizing them based on the effect on the program and components where the issues occurred. Different from these studies, this work investigates the symptoms of ML exceptions that developers post on Stack Overflow. Our work can complement existing work by helping developers and researchers understand the ML exception symptoms, their relations, and how they affect community support.

Studies on exception handling and stack traces. Several studies have explored the handling of exceptions within source code. Previous investigations have analyzed the practices and patterns of exception handling in the source code [39–44]. Nakshatri et al. [43] investigated the patterns of handling exceptions in Java projects hosted on GitHub and SourceForge. Their findings revealed that many developers employ Java's exception-handling mechanism primarily for logging purposes and retrieving stack traces. The top five operations identified in exception handling included categorizing log statements, and printing stack traces. This underscores the significance of examining the logging practices related to exception stack traces. De Padua et al. [44] revisited exception-handling practices by analyzing 16 open-source Java and C# libraries. Their study focused on the quantity and diversity of exceptions, sources of exceptions, and the strategies employed. They found that a small portion of exceptions is handled specifically, while the majority is addressed with a generic strategy. While these investigations concentrate on the practices of exception handling within the source code and their influence on program behaviors, we performed an analysis in the components of exception and their impacts without having access to the source code of projects.

Adrian et al. [45] conducts an empirical investigation on using stack traces by Eclipse developers and proves the significance of stack traces in debugging. The study indicates that up to 60 percent of bug reports that contain stack traces led to successful resolutions, and faults were typically found in the top 10 stack frames. In other words, they highlight the importance of encouraging bug reporters to provide stack traces. Emillie et al. [46] introduce *PYCEE*

(Python Compiler Error Enhancer) as an extension of Sublime Text IDE and analyze the challenges of understanding and interpreting error messages produced by Python compilers. *PYCEE* utilizes SO to offer enhanced and customized error messages, which programmers found helpful during an evaluation. Different from these studies that investigate the exception handling and stack traces of traditional software systems, this work focuses on the exceptions of ML applications and mine Stack Overflow data to understand how developers post the ML exception symptoms and their effect on community support.

Developers rely heavily on stack traces to find the root causes of issues; hence, much research has been put into the area [47–50]. Most similar to ours, Medeiros et al. [47] utilize correlation analysis to find and group similar crash reports and identify buggy files in the domain of web-based systems. Also, they assess the performance of the resulting technique in industrial contexts. In this study, we focus on stack traces on SO posts that are related to ML applications. Besides, Sui et al. [50] try to build call graphs based on the stack traces collected from the GitHub issue tracker and SO forums. All of the investigations in the paper consider the Java programming language; however, our study is based on the stack traces of ML applications written in Python.

CHAPTER 3 METHODOLOGY

3.1 Overview

Figure 3.1 provides an overview of our comprehensive research work, which aims to illuminate the causes and symptoms of ML application issues by mining relevant data from Stack Overflow (SO). Our investigation unfolds across distinct studies and chapters, with the initial focus dedicated to uncovering the root causes of ML exceptions in the first study. Subsequently, in the second study, our attention shifts towards identifying and understanding the symptoms associated with these ML exceptions.

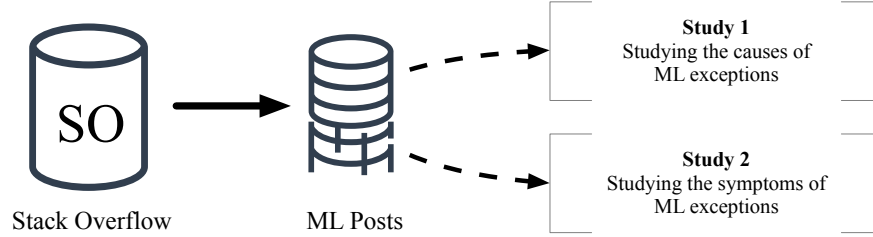


Figure 3.1 A general overview of our studies

3.2 Subject Data and Data Collection

Both studies concentrate on ML application exceptions using SO posts as the primary data source. In the first study, we extensively utilize the SOTorrent dataset [51], containing SO posts from July 31, 2008, to December 31, 2020, which includes metadata like question tags, view counts, scores, and acceptance status. Conversely, Study 2 updates the dataset using StackExchange Data Explorer, extending the analysis period to August 17, 2023. Both investigations employ SO posts to gain insights into ML application exceptions, conducting a comprehensive examination of the exceptions and related metadata for a thorough understanding of the subject.

In addition, our studies center on seven prominent ML libraries—namely, *TensorFlow*, *PyTorch*, *Scikit-learn*, *Keras*, *NLTK*, *HuggingFace* (or *Transformers*), and *Spark ML* (or *MLlib*). These libraries are selected based on their GitHub popularity metrics [52, 53], utilization in previous research [11, 54–56], and their implementation in the Python programming language. The focus is on these libraries due to their accessibility and efficiency in enabling ML developers to integrate ML solutions into their applications.

CHAPTER 4 STUDYING THE CAUSES OF ML EXCEPTIONS

Machine learning (ML), including deep learning, has recently gained tremendous popularity in a wide range of applications. However, like traditional software, ML applications are not immune to the bugs that result from programming errors. Explicit programming errors usually manifest through error messages and stack traces. These stack traces describe the chain of function calls that lead to an anomalous situation or exception. Indeed, these exceptions may cross the entire software stack (including applications and libraries). Thus, studying the ML-related patterns in stack traces can help practitioners and researchers understand the causes of exceptions in ML applications and the challenges faced by ML developers. To that end, we mine Stack Overflow (SO) and study 11,449 ML-related stack traces related to seven popular Python ML libraries.

First, we observe that ML questions that contain stack traces are less likely to get accepted answers than questions that don't, even though they gain more attention (i.e., more views and comments). Second, we observe that recurrent patterns exist in ML stack traces, even across different ML libraries, with a small portion of patterns covering many stack traces. Third, we derive five high-level categories and 25 low-level types from the stack trace patterns: most patterns are related to python basic syntax, model training, parallelization, data transformation, and subprocess invocation. Furthermore, the patterns related to subprocess invocation, external module execution, and remote API call are among the least likely to get accepted answers on SO. Our findings provide insights for researchers, ML library developers, and technical forum moderators to better support ML developers in writing error-free ML code. For example, future research can leverage the common patterns of stack traces to help ML developers locate solutions to problems similar to theirs or to identify experts who have experience solving similar patterns of problems. Researchers and ML library developers could prioritize efforts to help ML developers identify misuses of ML APIs, mismatches in data formats, and potential data/resource contentions so that ML developers can better avoid/fix model-related exception patterns, data-related exception patterns, and multi-process related exception patterns, respectively.

4.1 Introduction

The popularity of machine learning (ML) (including deep learning) applications has grown rapidly in recent years [1, 2]. Indeed, a 2020 Deloitte survey of 2,737 IT companies from nine countries found that modern machine learning technologies were being used by more

than 50% of these companies, and that 95% of the respondents were planning to use them within the following year [3]. Furthermore, in 2022, The NewVantage Partners’ *Data and AI Leadership Executive Survey* demonstrated that 91.7% of organizations were expanding their investment in data and AI activities, at the core of which is ML [2].

The growing trend in ML adoption delivers new challenges for ML developers (i.e., software developers and data scientists) [4–7]. When ML developers face issues in their development process, they can turn to technical question and answer (Q&A) forums such as Stack Overflow (SO) to answer their questions [5–9]. In this work, we analyze such questions to understand the failures (more specifically, exceptions) that technical forum moderators face when they develop ML programs. We target ML-related questions on SO because, with over 22 million questions, 33 million answers, and roughly 18 million users, SO is the largest technical Q&A forum [10]. A 2021 survey on SO [11] has shown that 80 percent of respondents visit the forum weekly, and 50 percent visit it daily. We, therefore, leverage SO to obtain data on exceptions that developers face when they develop ML programs because it is both popular and ingrained in developer work habits.

Prior works have studied the issues and challenges of developing machine learning applications [4–8, 12]. Indeed, it has been shown that the root causes of bugs and ML library challenges can be uncovered through a combination of bug reports and ML posts from GitHub and SO. However, the stack traces that manifest exceptions of ML applications remain unstudied.

A stack trace reports the chain of function calls at play when an anomalous situation, or exception, occurs. At the time of the exception, these function calls are either under execution or waiting for other function calls to be completed. The stack traces provide clues for understanding the errors that lead to exceptions (i.e., what chain of function calls lead to the exception). In this work, we study the stack traces of ML applications to understand the causes of exceptions in ML applications. We thus aim to provide insights to engage ML community support and leverage existing ML-related stack traces to improve current ML development practices. Our work provides a complementary perspective to existing works that study the issues and challenges of developing ML applications.

In this work, we study ~11K ML-related stack traces posted in SO questions, aiming to understand common patterns that cause exceptions in ML applications. To identify ML-related SO questions, we consider questions related to the use of seven popular Python ML libraries, including TensorFlow [13], Keras [14], Scikit-learn [15], PyTorch [16], NLTK [17], HuggingFace [18], and Spark ML [19]. We focus on the Python language as it is the dominant

programming language for ML application development [20,21]. We perform quantitative and qualitative analyses to understand the characteristics of SO questions with stack traces and to analyze the patterns of stack traces therein. Through our analyses of ML-related stack traces, we aim to provide insights for researchers, ML library developers, and technical forum moderators to help ML developers improve current ML development practices.

In particular, we structure our study and findings along the following three research questions (RQs):

- **RQ1. What are the characteristics of ML-related posts with stack traces?** To understand the significance of stack traces and their impact, we study the prevalence of SO questions with stack traces and how they impact community support on SO. We observe that 12.6% of posts related to our studied libraries include a stack trace. Although these posts receive more attention, they are less likely to have accepted answers, and take longer to be answered.
- **RQ2. What are the characteristics of the stack trace patterns in ML-related questions?** To understand the causes of ML application exceptions, we mine the common patterns in the ML stack traces posted on SO. We observe that most patterns are short (80% of the patterns contain fewer than five calls), and 19.53% are shared among at least two ML libraries. We also find that a small percentage of patterns (20%) span a large number of questions (up to 85%).
- **RQ3. What are the categories of the stack trace patterns, and which categories are most challenging for developers?** Through a qualitative study, we identify five high-level categories and 25 low-level types of stack trace patterns associated with ML development. We find that exceptions are often caused by misunderstandings or uncertainties related to ML API usage, data formats, or language constructs. Furthermore, exceptions related to external dependencies or manipulations of artifacts are less likely to receive timely community support.

The main contributions of our work include:

- This study is the first step towards understanding the stack traces of machine learning applications and their related issues posted on SO.
- We observe that questions with stack traces are less likely to get timely accepted answers. Technical forum (e.g., SO) moderators and researchers should pay particular attention to these questions to help developers.

- We observe patterns in stack traces of ML applications, and find that a small subset of them cover a wide range of questions, even across different libraries. Focusing on these prevalent patterns could produce impactful improvements.
- We provide a hierarchical taxonomy of the stack trace patterns of ML applications and recommend actions to improve ML applications and their usage.
- Our observations identify the most challenging pattern types and provide critical insight for future research.

4.2 Overview

Figure 4.1 gives an overview of our approach. We first extract ML-related posts (which contain questions and answers) from SO. Then, we extract stack traces from the posts (through regular expressions) and use pattern mining techniques to discover patterns in the stack traces. The extracted stack traces and their patterns are used to answer our research questions. In the rest of this section, we elaborate on each of these steps to precisely capture each step of our experiment setup. The detailed analysis of each research question is presented in Section 4.6.

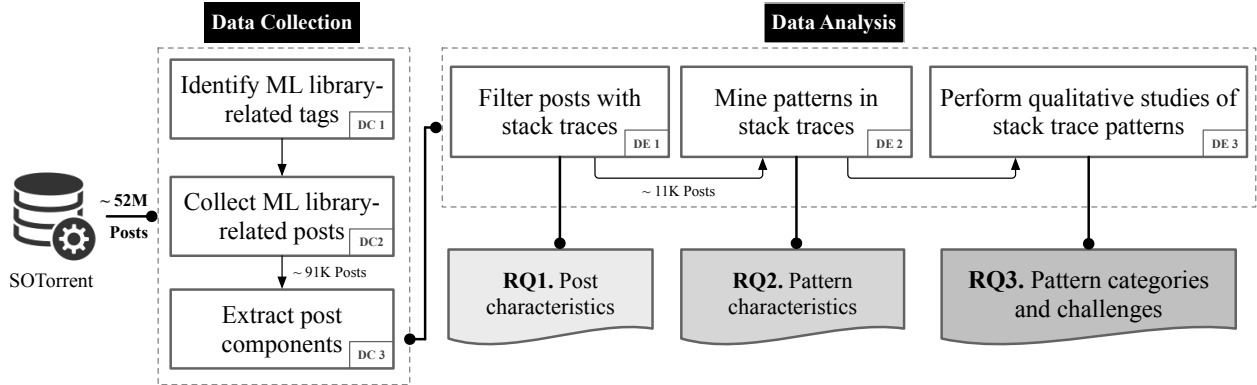


Figure 4.1 An overview of our study design

4.3 Subject Data

We use SO posts as our primary data source to study the stack traces of ML applications. Specifically, we use SOTorrent [51], an open dataset, to access the set of SO posts that contains questions, answers, and their metadata (e.g., question tags, view counts, scores, question and answer creation time, and whether an answer is accepted or not, etc). We use

the last released version of SOTorrent, released on 2020-12-31, and use Google BigQuery to access the dataset. The dataset contains the entire version history of all SO posts in the official SO data dump from its first post on July 31, 2008, until December 31, 2020.

4.4 Data Collection

[DC1] Identify ML library-related tags

We use SO question tags to identify ML-related posts. Using these tags, we focus on the questions related to the Python programming language and our selected ML libraries (e.g., TensorFlow). We choose the Python programming language because it is the most popular programming language for data scientists [11, 57] and involves libraries that are popular in the ML field [54–56]. To identify the tags related to Python ML libraries, we first define our studied libraries.

Defining studied ML libraries. We consider SO questions related to seven popular ML libraries: *TensorFlow* [13], *PyTorch* [16], *Scikit-learn* [15], *Keras* [14], *NLTK* [17], *Hugging-Face* [18], and *Spark ML* [19]. These popular and state-of-the-art libraries provide ready-made algorithms that allow ML developers to efficiently apply ML-based solutions in their applications. We select these seven ML libraries based on (1) their popularity metrics on GitHub [52, 53] (e.g., the number of forks, stars, and releases), as well as (2) their usage in previous studies [11, 54–56].

Identifying a tag-set. We leverage StackExchange¹ and SO tags² to find and extract tag names related to ML libraries. The StackExchange query space provides an environment to find all of the existing tags that match a specified pattern through the LIKE operator. For example, to find the tags related to the TensorFlow library, we use the query `TagName LIKE '%tensorflow%' THEN 'tensorflow'`. This finds all of the versions and subcategories related to TensorFlow like *tensorflow-lite* and *tensorflow2.x*. We use a combination of queries to obtain a relevant tag-set for each of our studied ML libraries. Our replication package [58] contains our queries and our complete tag-set.

[DC2] Collect ML library-related posts

To extract posts based on our tag-set, we use the Google Cloud Platform service (BigQuery). We export all post answers and ML-related questions using two queries. We use the output of

¹<https://data.stackexchange.com/stackoverflow/queries>

²<https://stackoverflow.com/tags>

these queries to extract the specific part of the metadata that we need to answer our research questions.

[DC3] Extract post components

Posts on SO can be composed of text, images, code snippets, and more. For simplicity, as shown in Figure 4.2, we summarize each post's body into text and code blocks. *Text Blocks* contain the questioner's textual descriptions, and *Code Blocks* are composed of code snippets or reports such as stack traces. In this study, we focus on posts that contain *Code Blocks* with stack traces. Although SO often uses the (`<pre><code>`) tag to distinguish the *Code Blocks*, there are cases where *Code Blocks* obey other tags. Thus, we apply a lightweight heuristic approach that uses multiple regular expressions to identify *Code Blocks* in the SO posts and automatically extract all *Code Blocks* and *Text Blocks*. Our heuristics, and all the regular expressions, are provided as scripts in our replication package [58].

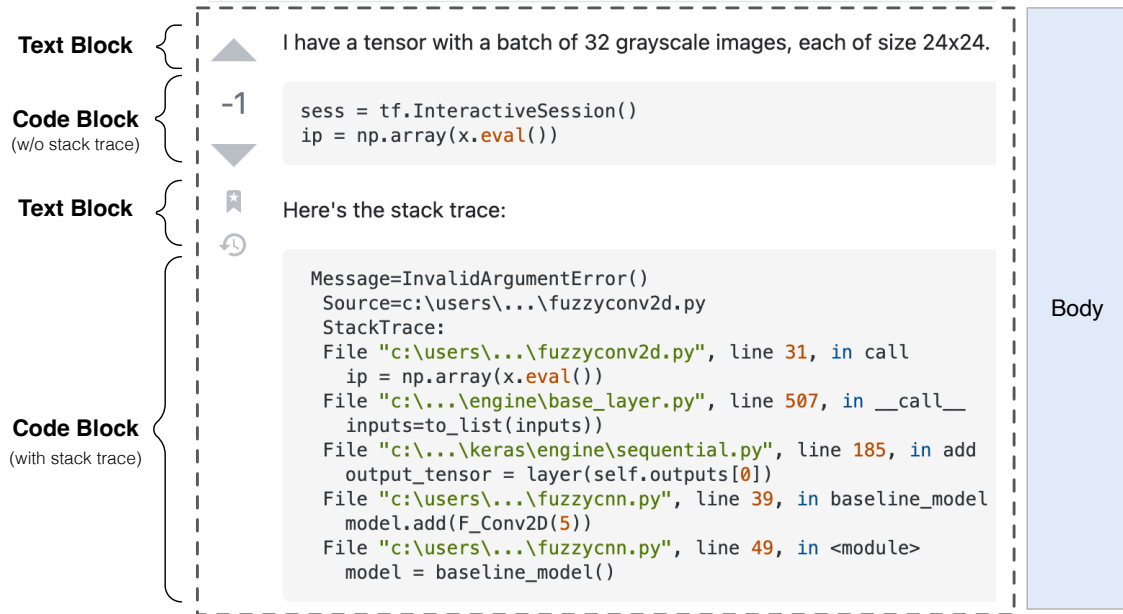


Figure 4.2 A truncated example of a SO question. (ID=52253378)

4.5 Data Analysis

[DE1] Filter posts with stack traces

Our research focuses on stack traces. Therefore we wish to distinguish between questions with *Code Blocks* with stack traces and other types of *Code Blocks* such as code snippets.

Regardless of operating system (OS) type, all stack traces contain OS paths. We use this knowledge to extract stack traces automatically through regular expressions.

[DE2] Mine patterns in stack traces

Stack trace transformation. The regular expressions used in DE1 provide us with two advantages: (1) we can differentiate between types of Code Blocks, and (2) we can capture the specific parts of each stack trace. Figure 4.3 indicates a stack trace recognized using our regular expressions. Moreover, this figure shows how we convert each stack trace to a list of pairs. Each pair contains the file’s and function’s names, which come from each line in the stack trace. The combination of file and function names, the yellow and red colored words in Figure 4.3, gives us unique pairs that we use to feed our pattern mining procedure.

Stack trace mining. In this study, we seek to uncover stack trace patterns and minimize the information lost due to the aggregation of different stack traces. For this purpose, we implement CC-Span [59], to mine closed contiguous sequential patterns. The algorithm considers two aspects: the items’ adjacency and the patterns’ closure. We use the unique pairs obtained from the previous step as input.

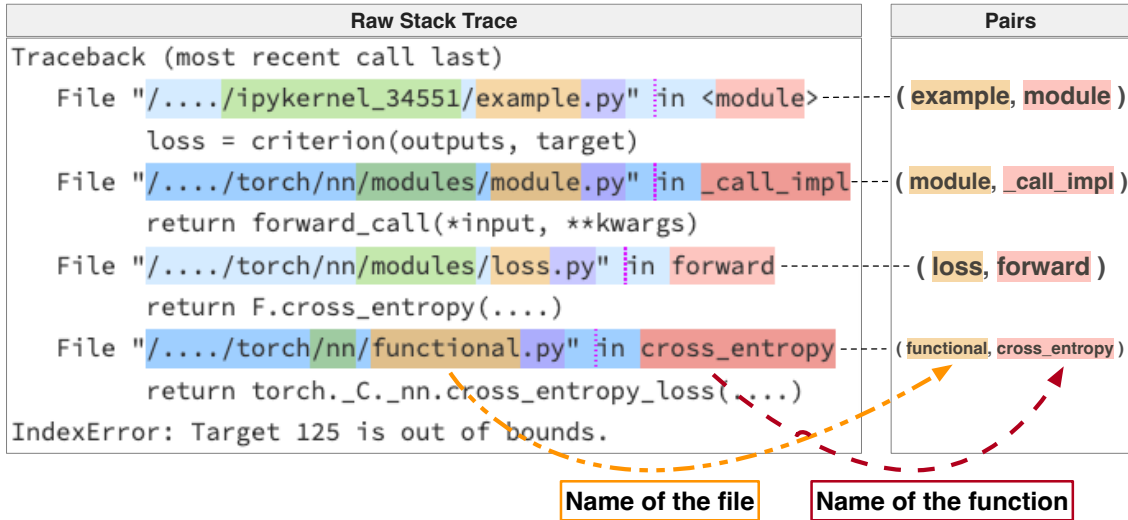


Figure 4.3 A truncated example of stack trace transformation. (ID=70716638)

[DE3] Perform qualitative studies of stack trace patterns

We performed qualitative studies to identify categories of stack trace patterns, to better understand the types of stack trace-related issues faced by ML developers and their challenges. Our qualitative studies are detailed in Section 4.6.

4.6 Research Questions and Results

(RQ1) What are the characteristics of ML-related posts with stack traces?

Motivation

When ML developers face challenges, they can post questions that contain stack traces on SO. These stack traces can indicate errors in their code, and communicate information about the challenges they face in their development process. Studying the characteristics of SO questions with stack traces can help ML developers and forum providers understand the profile of these questions and their level of community support. Therefore, this RQ proposes a quantitative analysis of the characteristics of ML-related SO questions with stack traces, and compares them statistically to SO questions without stack traces. This RQ also provides context for our next two RQs, where we aim to study the patterns in the stack traces of ML-related questions.

Approach

In this RQ, we exclude post answers to concentrate on the characteristics of SO questions with stack traces and compare them to those without stack traces. In particular, we study the questions' lengths, the questions' popularity (e.g., view count), and how likely these questions are to receive answers that are accepted by the questioners.

Studied characteristics of SO posts. Below we describe the characteristic variables that we study to determine the status of ML-related stack traces on SO.

- **Question Length:** measures the number of words used in all *Text Blocks* of a question.
- **Code Length:** measures the size of the *Code Blocks* of a question. We count the number of lines in each *Code Block* of a question. Our calculation ignores empty lines.
- **Question Score:** a measure of popularity. The question score shows the difference between the number of upvotes and the number of downvotes for a question [60].
- **Comment:** a measure of popularity. A count of comments for a given question.
- **View:** a measure of popularity. SO counts the number of views for every page load³. The total number of views for a given question.

³<https://meta.stackexchange.com/questions/36728/how-are-the-number-of-views-in-a-question-calculated>

- **Answer Count:** measures the count of proposed answers for a given question.
- **First Answer:** measures the hours between a question’s publication and when it receives its first answer.
- **Accepted Answer:** measures the hours between a question’s publication and when it receives an accepted answer. Question posters can choose one answer that they believe is the best answer to their questions.

Each characteristic is carefully selected to provide relevant data. Indeed, a question’s length can be a proxy for its complexity (longer questions are more likely to be complicated to describe). The length of code in a question can give an estimate of how technically challenging it is (questions with more code are likely more technically challenging) [61]. Evaluating question scores, views, number of answers, and comment counts depict community interest and attention [62]. Besides, the time it takes for the first and accepted answers highlights can provide insight into the ease at which the community can answer the question [63]. By using these factors we aim to obtain a complete picture of ML stack trace-related questions on SO.

We apply two statistical tests (i.e., Mann-Whitney U Test and Two-sample proportion Z Test) to compare these characteristics between questions with and without stack traces.

Mann–Whitney U Test. We apply this test [64] to assess whether the data for ML-related stack traces can be considered statistically independent from questions w/o stack traces. We chose a nonparametric test because we do not know the distribution of our data a priori. Moreover, we execute the test at the 5% significance level (i.e., $\alpha = 0.05$).

Two-sample proportion Z Test [65, 66]. We apply this test in Table 4.5, to compare the ratio of each ML library that gets accepted answers, for groups with and without stack traces. We use this statistical test because there is insufficient information to utilize the chi-squared distribution. We execute the test using a 5% significance level.

Results

An average of 12.6% of ML-related questions provide stack traces. While the number of questions fluctuates based on the ML library, the percentage of questions with stack traces is relatively stable. As seen in Table 4.1, TensorFlow (13.6%) has the most questions with stack traces, and NLTK has the least (10.0%). This shows that a number of developers see an inherent value in adding stack traces to their SO questions. Furthermore, based on the stack trace paths, we find that most SO questions that contain stack traces

Table 4.1 The number of questions on SO for each studied ML library, including all questions, those that contain *Code Blocks* and those that contain stack traces

ML Libraries	# Ques.	# Ques.		# Ques.		OS ¹	
		with code block		with stack trace		Unix-based	Windows
TensorFlow	39,690	32,368	(81.6%)	5,436	(13.6%)	69.1%	30.9%
Keras	21,568	18,506	(85.8%)	2,715	(12.5%)	70.8%	29.2%
Scikit-learn	18,133	14,795	(81.6%)	2,009	(11.0%)	62.0%	38.0%
PyTorch	5,540	4,620	(83.4%)	713	(12.8%)	76.2%	23.8%
NLTK	5,592	4,381	(78.3%)	564	(10.0%)	58.7%	41.3%
Hugging Face	236	204	(86.4%)	40	(16.9%)	87.5%	12.5%
Spark ML	141	126	(89.4%)	16	(11.3%)	93.8%	6.20%

¹ OS stands for Operating System.

come from Unix-based systems. This can be interpreted in two ways. Either most ML developers work and execute their code on Unix-based systems, or Unix-based developers are more likely to add a stack trace to their SO questions. Future work is needed to determine which is the case. However, in all cases, we find that stack traces are used in ML-related questions irrespective of the ML library in question. Table 4.2 presents the distribution of the stack traces on SO for each ML library. The table shows that questions with stack traces rarely (at most 6%) contain more than one stack trace.

Table 4.2 The distribution of stack traces on SO for our studied ML libraries

ML Libraries	# Stack Trace Instances	# Ques. (with stack trace)	Ratio
TensorFlow	5,728	5,436	1.05
Keras	2,851	2,715	1.05
Scikit-learn	2,090	2,009	1.04
PyTorch	741	713	1.03
NLTK	585	564	1.03
Hugging Face	40	40	1.00
Spark ML	17	16	1.06

Questions with stack traces have shorter natural language descriptions and longer *Code Blocks* compared to questions without stack traces. Table 4.3 displays the median text length and code length of questions with and without stack traces. Irrespective of library, questions with stack traces are less wordy (median 11 fewer words on average) but

Table 4.3 The median question and code lengths for the SO posts related to our studied ML libraries

ML Libraries	Ques. Length (Words)			Code Length (LOC)		
	with stack trace	w/o ¹ stack trace	Sig. ²	with stack trace	w/o stack trace	Sig.
TensorFlow	69.0	84.0	***	25.0	19.0	***
Keras	73.0	86.0	***	24.0	21.0	***
Scikit-learn	67.0	79.0	***	18.0	14.0	***
PyTorch	68.0	77.0	***	19.0	17.0	*
NLTK	61.0	72.0	***	15.0	11.0	***
Hugging Face	67.5	90.0	*	18.0	14.0	o
Spark ML	83.5	80.0	o	43.0	16.0	***

¹ w/o = without

² Statistical significance of explanatory power according to Mann–Whitney U Test:

o $p \geq 0.05$; * $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$

use more lines of code (median 5 more LOC on average) compared to questions without stack traces. Therefore, it appears that stack traces allow question posters to express their problem with fewer words by relying on the lines of code provided by the stack traces themselves.

Questions with stack traces gain more popularity (in terms of view counts and comment numbers) than questions without stack traces. Table 4.4 compares the status of community support, including score, comment, view, and answer count among ML-related questions with and without stack traces against a baseline comprised of all Stack Overflow questions. While questions with stack traces garner lower scores than questions without stack traces, the median difference is small (~ 1). In terms of comment count and view count, questions with stack traces get more, or an equivalent number of comments and views compared to questions without stack traces and the SO baseline. Finally, we find that the median answer count amongst different ML libraries is the same in all groups, with a median answer count of 1. Therefore, it appears that using stack traces in ML-library questions encourages comments and views.

Questions with stack traces are less likely to get answers and take longer to get them when they do. We use the Two-sample proportion Z Test to compare the ratio of questions that receive an accepted answer with and without stack traces. Using Table 4.5, we can see that questions with stack traces are less likely to receive an accepted answer. Indeed, for all statistically significant results, questions with stack traces take longer to obtain a first answer, and an accepted answer. While some libraries obtain answers faster than others, the trend holds across libraries, despite the overall answer speed. Indeed, Table 4.5 shows that questions with stack traces present unique characteristics that are distinct from questions

without stack traces, even from questions that may contain other *Code Blocks*, such as code snippets.

Summary of RQ1

SO questions with stack traces are prevalent across all studied libraries, with 12.6% of studied questions containing a stack trace. These questions generally use fewer words (a difference of 11 words on average) and garner more comments and views than questions without stack traces. However, they are less likely to have accepted answers, and take longer to be answered in the first place, despite the higher attention received. Forum moderators and researchers could pay particular attention to prioritizing such questions.

(RQ2) What are the characteristics of the stack trace patterns in ML-related questions?

Motivation

When a Python program faces an error or exception, the Python interpreter produces a stack trace that lists the function calls that lead up to the error. By examining the stack traces, we may identify patterns in the sequence of function calls that can help developers and researchers understand the root cause of the errors. In addition, finding frequent patterns among many stack traces may help us identify buggy functions and APIs for ML libraries. Finally, stack trace patterns can also help us identify code structure or design problems that may contribute to errors.

Approach

To answer RQ2, we first seek to identify stack trace patterns among ML-related questions. Transforming stack traces into lists of pairs of file and function names (see Figure 4.3) allows us to create a list of all stack trace instances for each ML library. We use these lists as input to the CC-Span algorithm [59] to uncover stack trace patterns. By grouping similar stack traces together and identifying patterns in the stack traces, we can gain a better understanding of the common issues and errors that occur when working with ML libraries.

The CC-Span algorithm uses a support threshold as an input parameter. Figures 4.4 and 4.5 display the output and the cumulative percentage of stack trace patterns for various supports and pattern lengths. These plots show that by setting the *support* = 2, on average,

we can enclose 83.88% and 64.73% of questions and stack trace instances in the stack trace patterns set. Thus, we set the support threshold to the smallest number (2) to reach the highest coverage. We ignore singleton instances ($support = 1$) because they are non-recurring and therefore not patterns. Table 4.6 depicts the percentages of stack trace instances and questions that were maintained by using the support threshold value of 2.

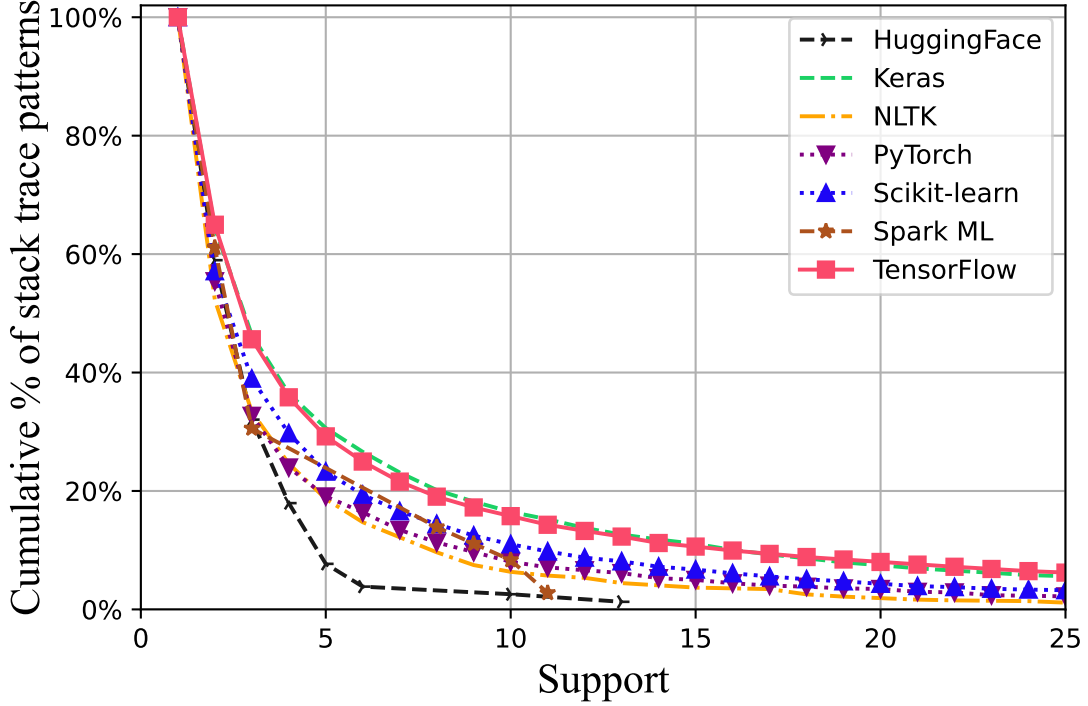


Figure 4.4 The cumulative percentage of stack trace patterns for various supports ($support \geq 1$)

Results

Recurrent patterns are common in the stack traces of ML-related questions. Table 4.6 reveals the percentage of stack trace instances and questions that fall into pattern sets with a support of at least 2, grouped by different ML libraries. Indeed, a high proportion of stack trace instances clearly have patterns, and on average, we cover 83.88% and 64.73% of total questions and stack traces, respectively. Furthermore, recurrent patterns are common.

A relatively small portion of patterns (20%) cover a large number of questions (75% to 85%). Figure 4.6 indicates the percentage of questions with stack traces that are covered by the percentage of stack trace patterns, for each ML library. For the majority of our studied libraries (i.e., not HuggingFace and NLTK libraries), 20 percent of significant

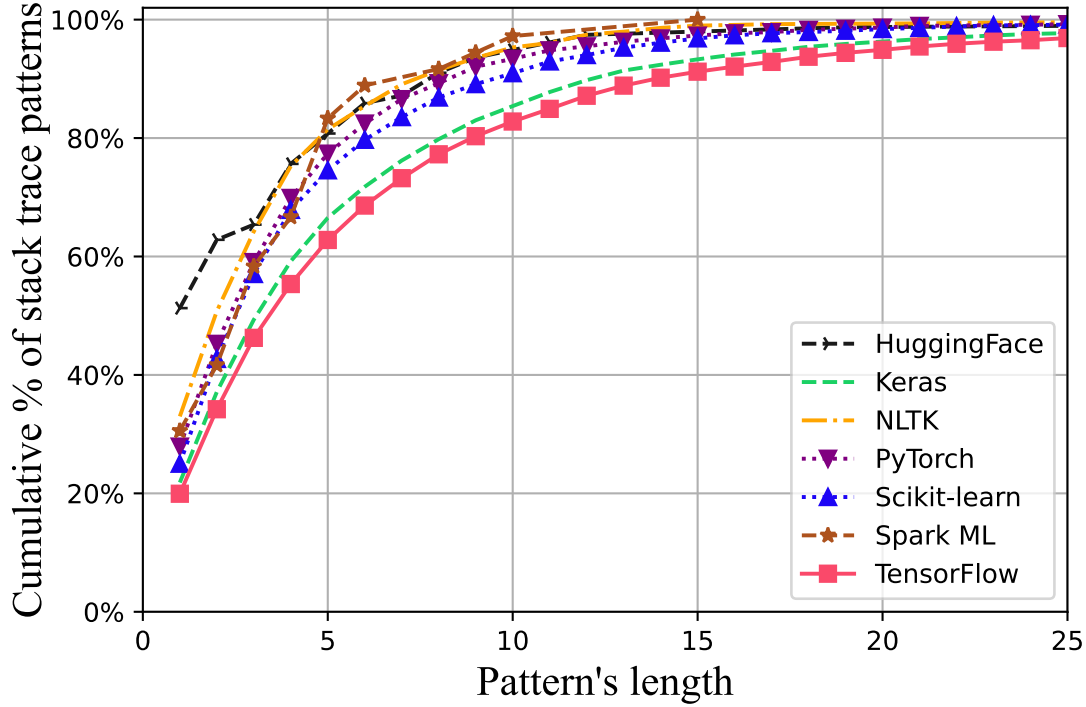


Figure 4.5 The cumulative percentage of stack trace patterns vs. pattern's length (*support* ≥ 1).

patterns (high-support patterns) constitute a considerable proportion of questions (75% to 85%) for each ML library, and the rest (80%) cover around 20% of all questions. As shown in Figure 4.4, for all ML libraries, many stack trace patterns (20%-65%) have low support (between 2 and 4); Less than 20% of stack patterns have a support equal to or greater than 10. The detailed distribution of the support values of each ML library's patterns can be found in our replication package [58]. Our results indicate the significance of studying these stack trace patterns as they cover the majority of the questions with stack traces.

Most patterns are composed of few calls (short pattern length), with 80% of patterns having a length shorter than or equal to 5. Figure 4.5 shows the percentage of stack traces found for different pattern lengths. Except for the Spark ML and HuggingFace libraries, 50% of stack trace patterns have a length between 1 and 3, and few patterns ($\sim 20\%$) are lengthy (more than 5 calls). This may indicate that many of the issues and errors encountered when working with ML libraries can be traced back to a relatively small number of specific function calls or code blocks. This provides opportunities to study these short patterns and identify their root causes. Also, understanding these common patterns can be helpful for developers who are seeking answers to their problems, as it can help them identify

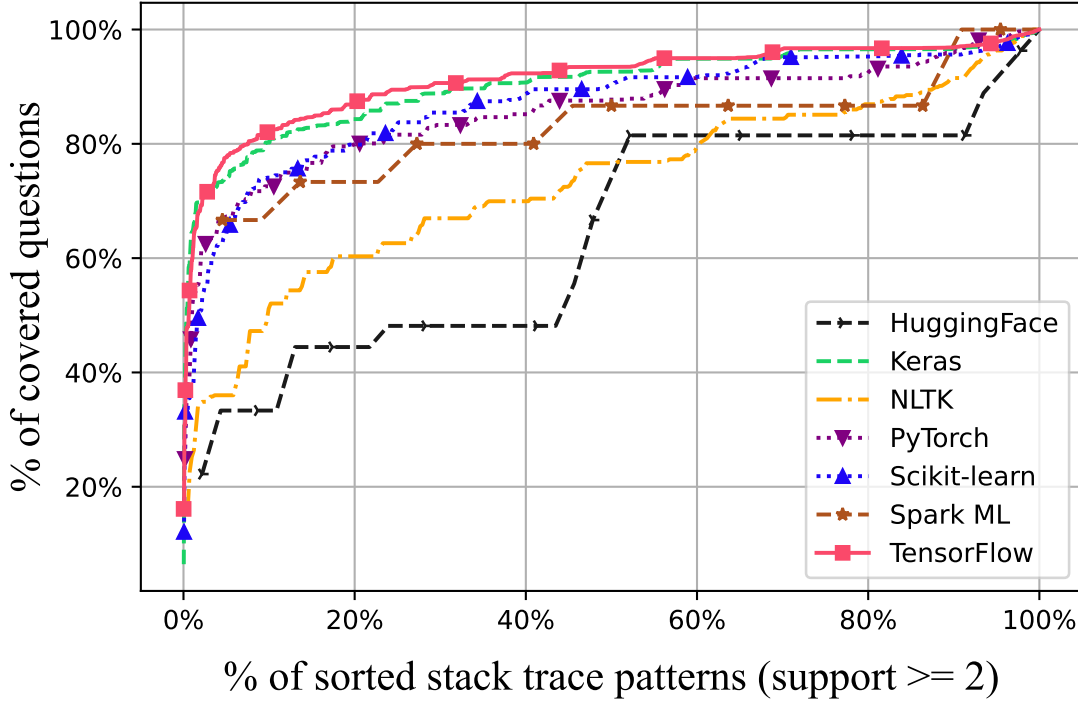


Figure 4.6 The percentage of covered questions based on stack trace patterns (patterns are sorted based on their supports).

error root causes and find solutions more quickly.

Stack trace patterns are shared across the questions of different ML libraries.

Figure 4.7 and Table 4.7 each give complementary insight into the number of patterns shared between ML libraries. Figure 4.7 shows the distribution of stack trace patterns for each ML library. Meanwhile, Table 4.7 shows the overall support for stack trace patterns.

Figure 4.7 is a heat-map plot that uses a warm-to-cool color spectrum to demonstrate the correlation between pairs of ML libraries using their shared stack trace patterns. Table 4.7 shows how often libraries share patterns. As can be seen in Table 4.7, few patterns are shared by many libraries. However, some pairs of libraries like TensorFlow and Keras have many (2,044) shared patterns. The Keras library leverages the underlying framework of TensorFlow for the execution of basic operations, its stack traces are therefore expected to be similar to TensorFlow's. Our approach aptly identifies this inherent correlation, an expected finding, which improves our confidence in the overall results.

Indeed, TensorFlow shares patterns with many other ML libraries. This could indicate that these libraries encounter similar issues and errors. Analyzing shared stack trace patterns can be useful for identifying common problems when working with these ML libraries. By

understanding these patterns, developers may anticipate and troubleshoot potential issues when working with these libraries, helping them find solutions more efficiently.

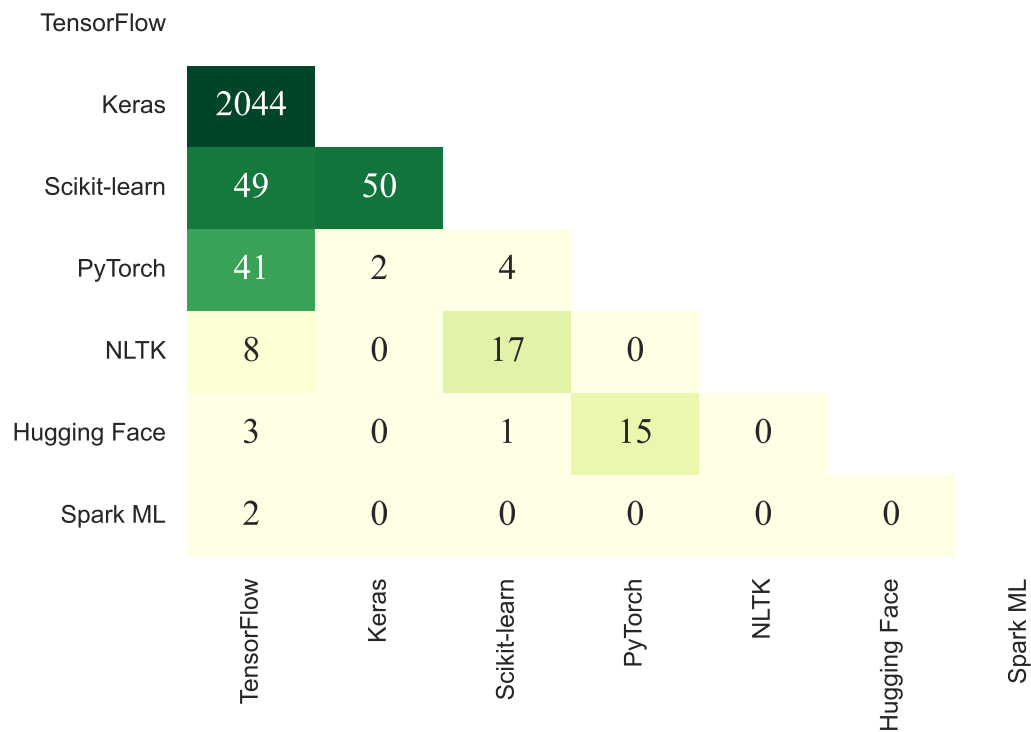


Figure 4.7 Stack trace pattern correlation between ML-libraries. The heatmap plot displays the extent of shared patterns, ranging from 0 (light green) to 2044 (dark green), based on the intensity of color.

Summary of RQ2

Recurrent patterns are common in the stack traces of ML-related questions, and a relatively small portion of these patterns cover many questions. Most patterns have a short length, and some patterns are shared across the questions of different ML libraries. Future research can leverage such common patterns to help ML developers locate solutions to problems similar to theirs or to identify experts who have experience solving similar patterns of problems.

(RQ3) What are the categories of stack trace patterns, and which categories are most challenging for developers?

Motivation

In the previous RQs, we observe that stack traces are prevalent in ML-related questions on SO (RQ1), and that these stack traces share common patterns (RQ2). To have a deeper understanding of the stack trace patterns, in this RQ, we perform a qualitative study to categorize them. We also analyze which categories of patterns receive less community support on SO. Our results can provide insights into the causes of exceptions in ML applications and shed light on future work to help practitioners overcome their challenges.

Approach

We perform our qualitative study in three rounds (i.e., Round 1, 2, and 3). In Round 1, we employ the stack trace patterns shared among ML libraries to cover a diversity of ML libraries. In Round 2, to consider the patterns’ impact and sample from all captured patterns, we purposefully sample the stack trace patterns to assign a higher sampling probability to the patterns with higher support. In the last round, Round 3, we derive stack trace categories. We explain each round in detail in the following subsections.

[Round 1 of our qualitative study]

In the first step, we look for shared patterns among ML-related libraries. Using Table 4.7, we choose patterns that are shared among 3, 4, 5, and 6 libraries (213 stack trace patterns). To understand the pattern types, we manually analyze these 213 patterns. We explain our first qualitative analysis process in detail below.

Qualitative analysis of shared patterns. We used grounded theory [67–70] to identify stack trace pattern categories. Grounded theory has three phases: open, axial, and selective coding and is appropriate for concluding a high-level abstraction from low-level definitions [70].

For each pattern instance, we randomly fetch five associated posts. We examine the file’s path, the function’s name, and the error line in each stack trace. Also, using the *Post ID* of each pattern instance, we manually check each post in its entirety, including its title, description, and code, both inside questions and answers. We use this information to understand the reason behind the error report.

We assign one type (label) for each pattern instance. If a stack trace pattern could belong to multiple types, we identify and choose its primary type. The author, supervisor, and co-supervisor jointly conduct the coding process by following the five phases listed below:

Step 1: Phase-1 coding. We randomly shuffle all 213 patterns and divide them into three parts. We distribute the first third of all patterns amongst three coders. Each coder labels (i.e., assigns a type to) them separately.

Step 2: Discussion after phase-1 coding. We discuss and analyze our resulting types and reach an agreement in order to obtain the same types for all coders.

Step 3: Revisiting phase-1 coding. According to the result of our discussion, we revise the phase-1 coding results to obtain consistent types.

Step 4: Phase-2 coding. In this phase, we again divide the remaining portion of the patterns (i.e., two-thirds) into three parts and assign each part to two different coders. Coders can still add new types during this phase.

Step 5: Discussion after phase-2 coding and resolving disagreements. We discuss and analyze types updated in phase-2 and reach a consensus. We finalize the types after this step. During this final evaluation, if we find a disagreement between two coders, the third coder resolves the disagreement for the particular pattern.

Measuring the reliability of Round-1 results. We calculate inter-rater reliability to measure the validity of our qualitative study results. The results of our qualitative studies are reliable if raters (or coders) reach a certain level of agreement on their labeling procedure.

Krippendorff's α [71, 72]. We use α to measure the agreement achieved after the coding process. α supports any number of coders as well as a standard coefficient for estimating inter-rater reliability. α is given by:

$$\alpha = 1 - \frac{D_0}{D_e} \quad (4.1)$$

where D_0 is the disagreement observed between multiple coders and D_e is the disagreement expected by chance. The α value ranges between 0 and 1, where 0 is the complete absence of agreement and 1 indicates perfect agreement between coders [71]. For this round (Round-1), we obtain a Krippendorff's α of 0.885, which indicates a reliable agreement (i.e., common understanding) between the coders.

[Round 2 of our qualitative study]

In our second qualitative study, we expand our investigations to all stack trace patterns instead of just shared ones (as done in Round 1). However, we obtain a total of 11,449 stack trace patterns, too many for a manual investigation. To select a reasonable number

of patterns, we focus our sampling on “critical patterns” by giving patterns with a higher support a proportionally higher chance to be sampled.

We use equation 4.2 to calculate how many patterns to sample, where A_i is the set of all of the stack trace patterns for library i . Meanwhile, n represents the total number of sampled stack trace patterns, and n_i represents the number of sampled patterns for library i .

$$\text{ceil} \left\lceil \frac{\text{len}(A_i) \times n}{\sum_{i=1}^7 \text{len}(A_i)} \right\rceil = n_i \quad (4.2)$$

To ensure that we choose at least one pattern for each ML library, we use Eq. 4.2 with $i = 1$, and $n_1 = 1$, to force a pattern to be selected from the library with the least patterns (i.e., Spark ML with 22 patterns), this implies $\text{len}(A_1) = 22$. Furthermore, we have a total of 11,449 patterns, which implies that $\sum_{i=1}^7 \text{len}(A_i) = 11,449$. Using these numbers we can calculate n . Because we use *ceil* to round our result, there is some flexibility in the result. We choose to use the highest possible result. This yields a total of 782 patterns (i.e., n). We then use this to calculate sample sizes for other libraries. Details are available in our replication package.

Using Eq. 4.2 and the types identified in Round 1, we sample 782 patterns for the second part of our qualitative study.

Qualitative analysis of selective patterns. We perform a hybrid coding approach and obey the same structure as the previous Round (i.e., randomly selecting five posts, analyzing the posts containing the patterns, etc.). We use types from Round 1 and extend them in Round 2. As the coders have established a reliable common understanding of the labels in Round 1, the author is the primary coder in Round 2. The author’s supervisor and co-supervisor joined the author to help resolve uncertain labels and discuss the coding results. The following steps are listed below:

Step 1: Phase-1 coding. One of the coders (the author) codes all 782 patterns based on the types which come from Round 1. New types are added if they do not exist in the past coding process.

Step 2: Discussion and Revisiting after Step-1 coding. We hold a meeting to analyze the first step’s results. This meeting lasts about two hours, intending to reach an agreement. Also, during this meeting, all coders talk about allocating final types for those patterns and types that are difficult to code for the first coder. Because of the common understanding observed in Round-1, only one coder independently codes all patterns for this round. Therefore, Krippendorff’s α is not calculated for this round.

[Round 3 of our qualitative study]

Defining stack trace pattern categories. We use Axial coding ⁴ to find an answer for the second part of RQ3. Axial coding is a method of analysis used in qualitative research to recognize patterns and relationships within data. It involves breaking the data into smaller units and assigning them codes or labels. To that end, all three coders hold a discussion meeting and define the categories. In this meeting, we print the stack trace pattern types onto cards and classify them into categories. Then we merge and split certain stacks of cards through a short discussion as the need arises. The resulting categories are discussed in the section below.

Results

We identified 25 stack trace pattern types across five high-level categories, including model-related patterns, data-related patterns, Python language syntax-related patterns, external dependence-related patterns, and multi-process-related patterns. In total, we manually analyzed 995 stack trace patterns. This sample covers 8.2% of all stack traces in our dataset. Table 4.8 shows the open coding results after all three rounds of our qualitative studies, including the pattern category, pattern type, and frequency of the pattern types in Round 1 and 2, separately. The frequency column (*Freq.*) shows the total number of patterns categorized into a specific pattern type, the frequency of the pattern type in each round, and the percentage of patterns of that type. The table is sorted based on the number of patterns for each pattern type. Overall, we categorize the 995 manually analyzed stack trace patterns into the 25 distinct pattern types defined in the description column of Table 4.8. An example of each pattern type can be found in the Example Post column. Some (9) patterns were not categorized during the coding process. These patterns follow uncommon stack trace structures; we therefore deliberately ignore them and do not show them in the result table.

Although the number of patterns and the sampling approaches used in both rounds of our qualitative study are different, as presented in Table 4.8, there are few differences in the pattern types found in both rounds. Indeed, in the second round, we only add four new pattern types, including *Package installation*, *Model conversion*, *Data loading*, and *Data conversion*. Finally, *Model copy* is the only pattern that exists in Round 1 but not in Round 2. In total, the share of new pattern type instances in Round 2 is less than 2.5% of our sample. We therefore believe that we reached type saturation and identified representative patterns. The identified stack trace pattern categories are discussed below.

⁴<https://delvetool.com/blog/axialcoding>

[MOD] Model-related patterns. Pattern types classified into this category are related to the models of ML algorithms. Specifically, the model-related pattern types include model training, hyperparameter tuning, model validation, model saving/loading, model conversion, and model copying. The first three types are related to building and validating the models, while the latter types are related to manipulating the model artifacts.

```

1  Traceback (most recent call last)
2  /.../def_function in __call__
3  /.../training_v2.py in fit
4  /.../training_v2.py in run_one_epoch
5  /.../training_v2_utils.py in execution_function
6  /.../def_function.py in __call__
7  /.../function.py in __call__
8  /.../function.py in _filtered_call
9  /.../function.py in _call_flat
10 /.../function.py in call
11 /.../execute.py in quick_execute
12 /.../six.py in raise_from

```

Listing 4.1 [MOD] Model patterns: truncated example (ID=57842734). The failure is related to a dimension mismatch in the model training process.

Model-related exceptions are often caused by misunderstanding the input requirements of the model APIs or failing to meet such requirements. ML algorithms are often implemented as libraries (e.g., TensorFlow) and used as black boxes by developers [73]. The behaviors of the ML algorithms are abstracted by the APIs provided by the libraries. However, developers may misunderstand the behaviors of the APIs (e.g., providing incorrect arguments or input data). Such misunderstanding or misuse may cause errors such as input type mismatches (e.g., *"Expected bool, got 1 of type 'int' instead."* in post #43604917⁵) or input data dimension mismatches (e.g., *"Shapes (3, 1) and (1, 3) are incompatible."* in post #63481755⁶). Listing 4.1 shows an example stack trace that belongs to the model-related pattern category. We assign the *model training/learning* pattern type to the pattern highlighted in yellow (Line 2-6) in Listing 4.1. We chose this label because the issue that triggers the stack trace is linked with a failure in the model's learning process caused by the shape mismatch of the logits and labels. **Library providers or researchers may help developers avoid or address such issues by providing examples for the**

⁵<https://stackoverflow.com/questions/43604917>

⁶<https://stackoverflow.com/questions/63481755>

APIs or identifying API misuses.

[DAT] Data-related patterns. Pattern types classified in this category focus on the saving, loading, conversion, creation, validation, transformation, and operation of data.

```

1  Traceback (most recent call last):
2    File ".../training.py" in _method_wrapper
3    File ".../training.py", in fit
4    File ".../data_adapter.py", in train_validation_split
5    File ".../nest.py", in map_structure
6    File ".../nest.py", in <listcomp>
7    File ".../data_adapter.py", in _split
8    File ".../ops.py", in convert_to_tensor_v2
9    File ".../ops.py", in convert_to_tensor
10   File ".../tensor_conversion_registry.py" in _default_conv[...]
11   File ".../constant_op.py", in constant
12   File ".../constant_op.py", in _constant_impl
13   File ".../constant_op.py" in convert_to_eager_tensor

```

Listing 4.2 [DAT] Data-related patterns: truncated example (ID=63106341). The failure is because TensorFlow couldn't convert a Numpy array into a tensor.

Misunderstanding the data types, data shapes, or the expected input format of an API is the main cause of data-related exceptions. Data handling (e.g., data cleaning or data transformation) is essential in developing machine learning applications [74]. Due to the complexity and sheer size of the data, the different input formats of ML algorithms, as well as the variety of data handling APIs (e.g., Numpy or Pandas APIs), developers have difficulties in creating, loading/saving, converting/transforming, or validating their datasets. We find that issues related to this category are often related to the misunderstanding of the data types/shapes or the expected input format of an API. For instance, as described in post #52832028⁷, the developer posted a stack trace with an error message *"ValueError: Tensor conversion requested dtype float32 for Tensor with dtype float64: 'Tensor'"*. This error occurs when converting a Numpy array into a tensor, as TensorFlow could not convert the *float64* format into the tensor format whereas the *float32* format was expected. Listing 4.2 shows a stack trace example of the data-related pattern category. We identify the pattern (highlighted in yellow (Line 13)) as a data conversion pattern because the root cause of the failure is related to converting one data type to another (i.e., Failed to convert a NumPy array to a Tensor). **Future research or development platforms could help developers**

⁷<https://stackoverflow.com/questions/52832028>

identify mismatches in data formats (e.g., by comparing the shape of an input dataset and the expected input data shape of an API).

[SYN] Python language syntax-related patterns. This category reveals failures related to the functionality of the Python programming language. Pattern types in this category include Argument validation, Syntax/attribute extraction, Method wrapper, Python basic syntax, and Object copy.

```

1  Traceback (most recent call last):
2    File "...\\mlp2.py", in <module>
3    File "...\\mlp2.py", in <listcomp>
4    File "...\\frame.py", in __getitem__
5    File "...\\base.py", in get_loc
6    File "...\\index.pyx", in pandas._libs.index.IndexEngine.get_loc
7    File "...\\index.pyx", in pandas._libs.index.IndexEngine.get_loc

```

Listing 4.3 [SYN] Syntax patterns: truncated example (ID=57717423). The failure is caused By feeding the DataFrame objects instead of Array or Numpy into the one method of the LIME library.

Syntax-based exceptions are often caused by the misunderstanding of Python basic syntax such as the type or structure of Python objects. Surprisingly, 14.47% of the manually studied patterns are related to Python basic syntax. For example, post #56622503⁸ describes an error caused by an incorrect use of the Python built-in module “random” when the built-in module was unintentionally overridden by a local file of the same name. This result may be explained by the fact that many ML developers do not have computer science or software engineering training [75]. In addition, misunderstanding of the type or structure of Python objects is another main cause of this category of stack trace patterns (e.g., as indicated by the Attribute extraction or Object copy stack trace patterns). Listing 4.3 shows an example of the syntax-related stack trace patterns (highlighted in yellow, Line 5) and assigns the *Syntax/attribute extraction* pattern type to the pattern. The failure was caused by feeding the wrong object type. It means the developer passed *DataFrame* objects instead of *List* or *NumPy* into one method of the LIME library. **We encourage ML developers to build a solid knowledge of the used programming language to develop their ML applications more effectively and with higher quality.**

[EXT] External dependence related patterns. The issues that cause these types of stack traces are related to external environments or outside of the code. This category is

⁸<https://stackoverflow.com/questions/56622503>

composed of: Remote API call, External module extraction, File operation, and Package installation.

```

1  Traceback (most recent call last):
2      File "...\\pywrap_tensorflow.py" in <module>
3      File "...\\pywrap_tensorflow_internal.py" in <module>
4      File "...\\pywrap_tensorflow_internal.py" in swig_import_helper
5      File "...\\imp.py" in load_module
6      File "...\\imp.py" in load_dynamic

```

Listing 4.4 [EXT] External dependence-related patterns: truncated example (ID=55912838). The failure is related to loading a TensorFlow package and its dependencies on Windows 10.

Uncertainties in external dependencies are one of the main causes of the exceptions of ML applications. ML applications often rely on external dependencies (e.g., external module/package, remote API call, etc.) to accomplish complex tasks. However, such dependencies may come with uncertainties. For example, an ML application that loads external data may fail when the external data becomes unavailable. A manual investigation revealed that many such issues were caused by timeout when executing the external dependencies. For example, in post #61149306, the exception was caused by timeout when loading online package data (i.e., *"socket.timeout: The read operation timed out"*). Listing 4.4 shows an example of the external dependence related stack trace patterns (Line 5-6, highlighted in yellow). We assign the *Package installation* pattern type to the example as the installation of a specific (incorrect) version of the TensorFlow package caused the error. **Future efforts may help developers mitigate the impact of such problems by pointing them to more robust alternative dependencies.**

[MLT] Multi-process related patterns. This category reveals failures related to Parallelization and Subprocess invocations. In this category, failures are related to deploying, creating, or executing threads or processes to handle tasks.

```

1  Traceback (most recent call last):
2      File "...\\for_device_data.py" in <module>
3      File "...\\for_device_data.py" in testPlt
4      File "...\\dbscan_.py" in fit
5      File "...\\dbscan_.py" in dbscan
6      File "...\\base.py" in radius_neighbors
7      File "...\\parallel.py" in __call__
8      File "...\\parallel.py" in dispatch_one_batch
9      File "...\\parallel.py" in _dispatch

```

```

10 File "...\_parallel_backends.py" in apply_async
11 File "...\_parallel_backends.py" in __init__
12 File "...parallel.py" in __call__
13 File "...parallel.py" in <listcomp>
14 File "...base.py" in _tree_query_radius_parallel_helper
15 File "...binary_tree.pxi" in sklearn.neighbors.ball[...]
16 File "...binary_tree.pxi" in sklearn.neighbors.ball[...]

```

Listing 4.5 [MLT] truncated example (ID=64116586). The failure is related to a request for too many parallel jobs.

Multi-process related exceptions in ML applications are often caused by data or resource contentions. ML processes (e.g., training processes) are usually computationally demanding, thus multi-processing can help in speeding up the processes. An example is the multi-threading⁹ feature of TensorFlow. However, multi-processing may lead to data or resource contentions that cause failures. For example, the post #56447556¹⁰ reports an exception that was triggered by a data contention: multiple processes tried to alter the same data (*"TypeError: can't pickle _thread.lock objects"*). This problem could be fixed by disabling multi-processing (e.g., setting the argument *use_multiprocessing=False*). Listing 4.5 shows an example of the multi-process related stack trace patterns (Line 7-12, highlighted in yellow). We assign the *Parallelization* pattern type to the example because the failure was caused due to the creation of a high number of parallel jobs that increased memory consumption until failure. **Future research could help developers identify potential data/resource contentions in their ML applications.**

ML exceptions related to external dependencies or manipulations of artifacts (e.g., model or data) are least likely to receive timely community support on SO, indicating their difficulties for developers. Table 4.9 illustrates the statistical information of the pattern types with regard to their received community attention and support. For each pattern type, the table shows the number of questions associated with the pattern type, the median number of views of the questions, the median time taken to receive an answer, the percentage of questions that receive an accepted answer, and the median time taken to receive an accepted answer. As shown in Table 4.9, questions with stack traces related to external dependencies (Subprocess invocation¹¹, External module execution,

⁹https://www.tensorflow.org/api_docs/python/tf/config/threading

¹⁰<https://stackoverflow.com/questions/56447556>

¹¹Although we group the pattern type "Subprocess invocation" in the "Multi-process" category, it is also related to external dependencies as it involves the invocation of a new and different process, as described in Table 4.8.

Remote API call, File operation, Package installation) or related to manipulations of artifacts (Data saving, Object copy, Model conversion, Data conversion) are least likely to receive timely community support. Indeed, they are the least likely to receive an accepted answer or take the longest time to receive an accepted answer, or both. Such observations indicate that these exception types are particularly difficult for developers. In particular, Model conversion is the most challenging pattern type, with only 21% of the questions receiving an accepted answer. Even when an answer is received, it takes a long time (i.e., a median of 71.46 hrs) to receive it.

The difficulties of the exceptions related to external dependencies (e.g., Remote API call) indicate that although the ML developer community on SO may be familiar with ML algorithms, pipelines, and frameworks, they may be less familiar with the hardware/software environment and dependencies that support the development of ML applications. For example, in the post #42223668¹², exception (i.e., *"NotFoundError: Key y_3 not found in checkpoint"*) was caused by the wrong OS path. Because the user saves the checkpoint inside the working directory but looks for that inside the root path. Indeed, these errors are closer to software engineering rather than machine learning. Experts with mixed backgrounds in machine learning and software engineering could help ML developers answer these questions.

The difficulties of the exceptions related to manipulations of artifacts may be explained by the large size and complex structure of the artifacts. Indeed, machine learning models are becoming increasingly large and complex and are typically treated as black boxes, which makes manipulating such models error-prone (e.g., Model conversion). Similarly, the large size and heterogeneous structure of their data make it challenging to perform manipulations (e.g., Data conversion or Data saving). For instance, post #59078406¹³ wants to convert a pre-trained model into the TensorFlow Lite format. The developer encounters a conversion failure (*"tensorflow.lite.python.convert.ConverterError: TOCO failed."*). This is due to the fact that the *tf.lite.TFLiteConverter* API supports a limited number of ops to be transformed. Future work may help ML developers alleviate such challenges by supporting developers in testing/debugging large and complex ML artifacts (e.g., by converting them into smaller or simplified versions for testing/debugging purposes).

Discussion

ML library providers should print actionable error messages when raising exceptions. Our manual analysis revealed that many exception error messages lack actionable

¹²<https://stackoverflow.com/questions/42223668>

¹³<https://stackoverflow.com/questions/59078406>

information for the users. For example, in SO post #64273829¹⁴, a developer posted a stack trace with an error message *"KeyError: <ExtractMethod.NO_EXTRACT: 1>"* while trying to use TensorFlow Datasets¹⁵ to load the CelebA dataset. The problem was caused by a Google Drive quota limit (as indicated in the accepted answer). The developer was unable to figure out the root cause given the stack trace and the error message. The issue could have been resolved more easily (perhaps by the developer itself) with a more informative error message.

ML library providers could allow developers to access the templates of the error messages to aid developers in finding relevant forum posts. It is known that developers typically search for their issues before posting new questions on SO [76]. However, as observed in our manual analysis and prior work [77], duplicate posts are common, showing that developers sometimes can not find the answer to their question although a similar question exists. This may be due to the fact that the error messages printed by the libraries often contain dynamic information that is specific to the particular problem faced by a developer. For example, in the error message *"InvalidArgumentError : Incompatible shapes: [32,784] vs. [32,2352]"*, *"[32,784]"* and *"[32,2352]"* are dynamic information. If a developer uses the error message directly to search for similar posts, they may miss posts describing the same problem with different dynamic information (e.g., a different data shape). To mitigate this problem, library providers could help developers by giving access to error templates (e.g., *"InvalidArgumentError : Incompatible shapes: [NUM, NUM] vs. [NUM, NUM]"*). Library providers or development platforms could provide a convenient way to copy the templates (e.g., by providing options to copy the original error message or the template).

ML APIs should improve their input argument type and data shape validation. In our manual analysis, we found that, for some stack trace patterns, when incorrect data is sent to an API, many internal method calls can be traversed before an error is raised. For example, in post #39321495¹⁶, a developer asks a question about the *"AttributeError: 'list' object has no attribute 'isdigit'"* error. The *pos_tag()* function expects a *Series* variable as its input but receives a *List*. Thus, we recommend that ML APIs improve their input argument validation to reduce the spread of erroneous behaviors which can obfuscate the original root cause.

ML library providers could incorporate version information in their printed stack traces. In our manual analysis, we found many cases in which there was a mismatch between

¹⁴<https://stackoverflow.com/questions/64273829>

¹⁵<https://www.tensorflow.org/datasets>

¹⁶<https://stackoverflow.com/questions/39321495>

the code that the user used and the version of libraries they were attempting to use. For example, in post #64771558¹⁷, the developer faced an error (*"TypeError: Unexpected keyword argument passed to optimizer: learning_rate"*) when attempting to convert a Keras model into a Core ML model. The issue was resolved eventually by “using the same version of Keras and TensorFlow in the environment creating the model and the environment converting the model”. If the version information was indicated in the stack trace (e.g., by printing the version information of the libraries that raise the exceptions), developers might be able to identify the version mismatches with less effort. Thus, we recommend ML library providers identify ways to infer or store the signature and versions of APIs that generate stack traces. This information can be used to quickly understand which API belongs to which version of the library and more rapidly determine the root cause of errors. On the other hand, developers are recommended to carefully verify the versions of the libraries they are using.

Summary of RQ3

We derived five categories of stack trace patterns that are related to models, data, Python syntax, external dependencies, and multi-processing. These exceptions are typically caused by the misunderstanding of ML API usages, data format, or language constructs, or caused by the uncertainties in external dependencies or data/resource contentions in multi-process executions. Among them, the exceptions related to external dependencies or manipulations of artifacts (e.g., model or data) are least likely to receive timely community support on SO. Our observations shed light on future efforts from researchers and ML library developers to support ML developers better, for example, to identify misuses of ML APIs (for model-related exception patterns), mismatches in data formats (for data-related exception patterns), or potential data/resource contentions (for multi-process related patterns).

4.7 Threats to Validity

We identify the following threats to the validity of this study:

External validity threats. We select SO based on its popularity, importance, and related works. However, we could increase the generalizability by adding other sources to our research, such as GitHub, for future works. Also, in both studies, we focus on posts related to 7 ML-related libraries. We could further improve generalizability by examining a larger set. In addition, in the first study, the number of stack trace patterns chosen for Rounds 1 and 2 of our qualitative study may impact its generalizability. To minimize this impact,

¹⁷<https://stackoverflow.com/questions/64771558>

we select 995 stack trace patterns using different approaches, such as shared libraries and selective pattern approaches. We assume that our conclusions derived from these 995 stack trace patterns represent a starting point for the categories and challenges of the stack trace patterns faced by ML developers.

This study specifically concentrates on stack traces linked to machine learning. Nevertheless, future endeavors might broaden our investigation to encompass non-machine learning issues.

For the first study, we use the latest version of SOTorrent, which is based on two-year-old data. However, during our quantitative analysis’ second phase, we augment this data with newer data obtained using our own scripts. Interestingly, the results were quite similar, with no significant changes observed. Thus, we believe that our findings are applicable despite the age of the original data. For the second study, by updating the dataset with StackExchange Data Explore, we mitigate this concern.

While previous studies like [78] and [79], selected data based on post popularity, in our analysis, we did not filter out any posts based on their scores and quality. This allowed us to gather more ML exceptions. However, it is possible that some of our extracted patterns, for the first study, might exist in less popular posts. We believe that this is an asset because it allows us to explore how all posts make use of stack traces without discrimination. Thus, our insights can apply to a wider range of posters.

Internal validity threats: For the first study, the stack traces and patterns we use might not wholly indicate developers’ actual challenges. For example, a failure may trigger a stack trace, but the root cause of that failure may be hidden from the stack trace. Also, developers can truncate each stack trace before posting on SO. This concern is more likely to affect patterns that belong to the ML libraries with few stack traces, while patterns with high support can mitigate it.

Construct validity threats. The concordance between SO and SOTorrent is unclear. However, many prior works make use of and draw reasonable conclusions from SOTorrent. In addition, while we use the latest SOTorrent version, it dates from 2020-12-31; at that time, the popularity of some ML libraries was as high as it is currently. Therefore our results may be underestimates. Also, because of the high volume of input data, we cannot check all results individually. Although we perform manual inspections, it is still possible that we may have missed some rare cases that are hard to recognize. Thus, we provide the source code and input data related to this study in our replication packages [58,80].

Table 4.4 The median community support variables for SO posts of our studied ML libraries

ML Libraries	Ques. Score			Comment			View			Answer Count		
	with stack trace	w/o ¹ stack trace	Sig. ²	with stack trace	w/o stack trace	Sig.	with stack trace	w/o stack trace	Sig.	with stack trace	w/o stack trace	Sig.
TensorFlow	0	1	***	1	0	***	324.0	202.0	***	1	1	**
Keras	1	0	o	1	1	**	282.0	162.0	***	1	1	o
Scikit-learn	1	1	o	2	1	***	562.0	360.5	***	1	1	***
PyTorch	0	1	**	0	0	*	345.0	168.0	***	1	1	o
NLTK	1	1	*	2	1	***	651.5	394.0	***	1	1	o
Hugging Face	0	0	o	2	0	**	134.0	131.5	o	1	1	o
Spark ML	1.5	1	o	0	0	o	597.5	651.0	o	1	1	o
Baseline	1	-	-	1	1	-	294	-	-	1	-	-

¹ w/o = without

² Statistical significance of explanatory power according to Mann-Whitney U Test:

o $p \geq 0.05$; * $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$

Table 4.5 The median time to answer for SO posts of our studied ML libraries

ML Libraries	First Answer (Hour)			Accepted Answer (Hour)					Sig. ³
	with stack trace	w/o ¹ stack trace	Sig. ²	with stack trace	w/o stack trace	stack trace	ratio (with stack trace)	ratio (w/o stack trace)	
TensorFlow	5.82	1.88	***	3.96	2.40	***	0.33	0.39	***
Keras	3.27	1.86	***	2.24	1.84	**	0.36	0.39	***
Scikit-learn	2.97	1.53	***	2.50	1.56	***	0.41	0.48	***
PyTorch	4.10	2.53	***	3.08	2.53	o	0.38	0.42	***
NLTK	2.39	1.14	***	2.91	1.16	***	0.40	0.50	***
Hugging Face	21.24	18.48	o	0.98	14.10	o	0.27	0.42	o
Spark ML	2.40	2.43	o	1.97	1.79	o	0.50	0.49	o

¹ w/o = without
– Statistical significance of explanatory power according to Mann–Whitney U Test² and Two-sample proportion Z Test³.
o $p \geq 0.05$; * $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$

Table 4.6 What percent of questions and stack traces fall into the pattern set (*support* ≥ 2)

ML Libraries	Covered Ques. (%)	Covered ST ¹ (%)
TensorFlow	91.45	64.96
Keras	89.09	64.43
Scikit-learn	85.39	57.10
PyTorch	82.44	55.42
NLTK	77.58	52.28
Hugging Face	67.50	58.97
Spark ML	93.75	100
Overall	83.88	64.73

¹ ST stands for Stack Trace.

Table 4.7 The number of shared stack trace patterns amongst various ML libraries (*support* ≥ 2)

Shared Libraries (#)	2	3	4	5	6
Stack trace Patterns (#)	2,236	145	47	20	1

Table 4.8 Classification of 995 stack trace patterns

[Category] ¹ Pattern Type	Description	Example Post ²	Freq. ³		
			Total	(R1, R2) ⁴	Perc.
[SYN] Python basic syntax	This type describes issues with the use of basic syntax in the Python programming language.	56622503	125	(44, 91)	14.47%
[MOD] Model training/learning ⁵	This type of taxonomy covers patterns related to the learning pipeline of an ML model.	19938587	122	(11, 115)	14.12%
[MLT] Parallelization	This type indicates patterns about deploying a process across multiple threads or processors.	60905801	81	(23, 64)	9.38%
[DAT] Data transformation	This type deals with problems related to the wrong shape, type, or format of the data.	47499284	65	(8, 59)	7.52%
[MLT] Subprocess invocation	This type includes patterns for executing a new and different process to handle a new task.	44523492	58	(23, 42)	6.71%
[DAT] Data operation	Patterns come from faulty action manifests during some operations on variables	44725860	52	(1, 52)	6.02%
[EXT] External module execution	This type represents patterns related to locating and running External Python modules.	60705802	51	(25, 33)	5.90%
[SYN] Method wrapper	This type includes patterns related to wrapping functions to extend their functionalities.	60690327	49	(10, 45)	5.67%
[MOD] Model saving/loading	This type illustrates patterns about loading and saving ML models.	59757217	45	(2, 44)	5.21%
[SYN] Syntax/attribute extraction	This type aggregates patterns relevant to extracting an object's attributes.	37733245	40	(14, 30)	4.63%
[MOD] Model training/construction	This type considers patterns affecting building a model during the training process.	63624390	38	(1, 37)	4.40%
[DAT] Data saving/pickle	This type categorizes patterns w.r.t saving a python object as a binary file.	61916110	20	(14, 10)	2.31%
[DAT] Data validation	This type includes patterns that are related to the test or validation of a variable.	63939783	19	(3, 19)	2.20%
[MOD] Model validation	It includes patterns according to the testing and validation of a (trained) model.	43925714	18	(4, 15)	2.08%
[EXT] Remote API call	This type indicates patterns about establishing and opening HTTP connections.	59081201	14	(6, 8)	1.62%
[SYN] Object copy	This type is for constructing a new compound object.	44591282	14	(2, 13)	1.62%
[EXT] File operation	This type is about accessing and manipulating files in an operating system.	57095189	11	(4, 8)	1.27%
[EXT] Package installation	This type is about package installation and uninstallation	38541233	11	(8, 4)	1.27%
[DAT] Data creation	It includes the issues arising during the construction of a data object.	47585817	11	(-, 11)	1.27%
[MOD] Model conversion	Problems come up while converting a model to a new one.	53596521	5	(-, 5)	0.58%
[MOD] Model copy	This type presents patterns for cloning a model instance.	54584393	4	(4, -)	0.46%
[MOD] Hyperparameter tuning	This type covers patterns about model optimization.	62205174	4	(1, 3)	0.46%
[SYN] Argument validation	This type is about checking and meeting the conditions of the input argument of a function.	63811127	3	(1, 2)	0.35%
[DAT] Data conversion	Convert the type or structure of a data to a new one.	64327397	2	(-, 2)	0.23%
[DAT] Data loading	This type comprises patterns connected to the decoding and loading of a data object.	48911850	2	(-, 2)	0.23%

¹ We categorize patterns into 5 different categories, and to save space, we use an abbreviation form for each category. Pattern types that include in one category have the same abbreviation. Categories are: [MOD] Model-related patterns, [DAT] Data-related patterns, [SYN] Python language syntax-related patterns, [EXT] External depending related patterns, and [MLT] Multi-process related patterns.

² For more details about each pattern type, the readers can refer to its web link, "<http://stackoverflow.com/questions/>", followed by the post ID.

(e.g., the example link for the first pattern type is "<http://stackoverflow.com/questions/56622503>".)

³ The frequency column consists of (1) Total: the total frequency of each label for both rounds, (2) (R1, R2): displays the frequency of labels for every round, and (3) Percentage: the share of each label. In the total number, we ignore patterns that are common between rounds.

⁴ R1 and R2 stand for Round 1 and Round 2

⁵ Each pattern type may cover distinct aspects. By the slash symbol, we tried to indicate or bold one part of that.

Table 4.9 Statistical information of pattern types

[Category] ¹ Pattern Type	# Ques.	View ²	FAD ¹ (Hour) ²	AAR ¹ (%) ³	AAD ¹ (Hour) ²
[SYN] Python basic syntax	1,815	423	4.72	30%	4.46
[MOD] Model training/learning	1,733	350	4.74	39%	3.51
[MLT] Parallelization	1,082	480	5.40	38%	4.67
[DAT] Data transformation	670	252.5	3.43	38%	2.73
[MLT] Subprocess invocation	542	346.5	21.11	• 28%	23.74
[DAT] Data operation	878	356.5	9.11	36%	5.90
[EXT] External module execution	435	269	9.01	• 26%	13.05
[SYN] Method wrapper	1,286	256	4.48	32%	3.88
[MOD] Model saving/loading	313	359	9.62	31%	6.08
[SYN] Syntax/attribute extraction	305	206	9.58	32%	7.59
[MOD] Model training/construction	723	508	6.46	41%	4.17
[DAT] Data saving/pickle	20	266	12.33	40%	44.28
[DAT] Data validation	442	435	2.31	39%	2.53
[MOD] Model validation	116	599	2.06	40%	1.79
[EXT] Remote API call	34	276.5	3.36	• 26%	1.61
[SYN] Object copy	22	603	79.78	32%	127.64
[EXT] File operation	17	394	2.48	• 28%	0.52
[EXT] Package installation	30	1,312	1.60	47%	76.22
[DAT] Data creation	197	531	6.91	40%	4.62
[MOD] Model conversion	29	214	72.11	• 21%	71.46
[MOD] Hyperparameter tuning	19	381	20.69	42%	13.15
[SYN] Argument validation	38	1,018	2.34	37%	2.34
[DAT] Data conversion	46	105	23.22	• 20%	11.01
[DAT] Data loading	9	229	6.56	44%	10.56
Overall⁴	<u>251</u>	<u>357.75</u>	<u>6.51</u>	36.5	<u>5.285</u>

1 Some column headers are abbreviated due to space constraints. *FAD = First Answer Duration, *AAR = Accepted Answer Rate, *AAD = Accepted Answer Duration

2 The *View*, *First Answer Duration (FAD)*, and *Accepted Answer Duration (AAD)* columns display the median numbers of each pattern type.

3 The *Accepted Answer Ratio (AAR)* column shows the percentage of the total number of questions that received the accepted answer in each pattern type.

4 The median of medians

CHAPTER 5 STUDYING THE SYMPTOMS OF ML EXCEPTIONS

We examine the reported symptoms of exceptions and assess their impact on community support in resolving failures. Similar to the initial phase of the investigation, this study uses again the seven ML libraries on SO. However, in contrast to the first study, our focus extends beyond call stacks to contain additional aspects of exceptions, specifically the exception type and error message. This second study reveals the distribution of ML exception symptoms on Stack Overflow, highlighting that approximately half of the questions involve all three symptoms, while others feature specific combinations. The exploration of symptom ambiguity indicates that individual error message templates or exception types do not definitively identify ML exception-related questions. Notably, questions solely featuring call stack symptoms are less likely to receive prompt community support. However, incorporating exception types and error messages alongside call stacks enhances the likelihood and speed of obtaining accepted answers.

5.1 Introduction

Artificial Intelligence (AI) encompasses multiple interconnected domains, including machine learning (ML), deep learning, computer vision, and natural language processing [81]. Within software systems, ML-based applications contain at least one ML component [82]. A software component is defined as a module or unit, separate either logically or functionally from others [83]. Essentially, an ML-based application comprises both ML and non-ML components collaboratively functioning within a software system.

The surge in popularity of ML applications is evident in recent years, with a report from SO indicating that, as of March 2023, 70% of developers are either already operating or planning to integrate AI into their development processes [84]. This growing adoption of ML presents new difficulties for developers. Unfortunately, developers frequently encounter issues characterized by confusing or unhelpful failure descriptions [85]. In response to such challenges, developers commonly resort to seeking assistance through online Q&A forums [86] such as *Stack Overflow* (SO).

SO stands as one of the premier Q&A online platforms for developers [87]. A survey of nearly 65,000 developers conducted by SO in February 2020, revealed that 90% of respondents turned to SO when confronted with coding problems [84]. This highlights the pivotal role that SO plays in software development. Notably, our research concentrates on ML-related

questions within the SO domain to understand the failures (more specifically, exceptions) that ML application developers face when developing ML applications.

Prior works have indeed studied ML posts on forums [22–25], bugs in ML applications [31–38], and exception handling such as [39–46]. However, in contrast to these investigations, our study specifically focuses on SO posts related to ML exceptions, providing a detailed analysis of elements of exceptions for ML-based applications.

Our study focuses on exceptions in ML applications, specifically examining the symptoms that include Call Stack (CS), Exception Type (Ext), and Error Message (EM). These symptoms play main roles in offering comprehensive insights into encountered failures. The Call Stack reveals the sequence of function calls leading to the failure, the Exception Type specifies the nature of the failure, and the Error Message provides specific details about the issue. By focusing on these symptoms, our objective is to not only identify ML exceptions but also gain a deeper understanding of the underlying problems faced by ML-based applications.

In this study, we study ~26K ML-related exceptions posted on SO, aiming to find their symptoms and impacts on community support in ML applications. To determine SO questions specifically related to the ML topics, we narrow down our focus to questions linked with seven well-known Python ML libraries: TensorFlow [13], Keras [14], Scikit-learn [15], PyTorch [16], NLTK [17], HuggingFace [18], and Spark ML [19]. We concentrate on the Python programming language, given its predominant role in developing ML applications [20,21]. By examining ML-related exceptions, our goal is to provide valuable insights for researchers and technical forum moderators, aiding ML developers in enhancing their current development practices.

In particular, we structure our study and findings along the following three research questions (RQs):

- **RQ1. What are the distributions of ML exception symptoms posted on Stack Overflow?** By dividing stack traces into distinct exception symptoms, we aim to comprehensively understand each symptom’s contribution to identifying ML exceptions. Furthermore, we aim to explore whether insights gained from understanding the symptoms of one exception can be extrapolated to locate other similar exceptions. We observe that approximately 50% of ML-related questions involving exceptions encompass all three symptoms: call stacks, exception types, and error messages. The remaining questions typically feature either a call stack alone or a combination of an exception type and an error message.
- **RQ2. How ambiguous are the different symptoms of ML exceptions?** Since

SO posts can choose to present either all exception symptoms, one exception symptom, or indeed any combination of symptoms, it is unclear how different exception symptoms help to understand ML exceptions unambiguously, and indeed much information can contribute to their definitive identification. We observe that posting error message templates or exception types individually cannot unambiguously identify exception-related questions. Similarly, an exception type or an error message template can be associated with completely different call stacks.

- **RQ3. How do the exception symptoms impact community support?** To understand the impact of different exception symptoms on community support. We observe that ML questions with all three exception symptoms are not the most likely to receive timely community support.

The data and scripts used to produce this study are shared in a replication package [80]

5.2 Overview

Figure 5.1 provides an overview of our approach. First, we define the desired tags (based on the ML libraries). Then, we extract all ML-related posts containing questions and answers from two sources (SOTorrent and StackExchange). In the next step, regular expressions are used to extract components inside the questions, such as call stack, exception types, and error messages. We then process the extracted components of the posts to answer our research questions. In the following parts of the section, we provide a detailed explanation of each stage of our experimental setup. The detailed analyses for answering each research question are presented in Section 4.6.

5.3 Subject Data

Our main data source for analyzing ML-based application exceptions is gathered from open forum posts. More specifically, we rely on SOTorrent [51] and StackExchange¹. SOTorrent is a dataset that collects Stack Overflow (SO) posts, including questions, answers, and their metadata, such as question tags, view counts, scores, creation times of questions and answers, and acceptance status. This dataset includes the complete chronology of all SO posts, spanning from July 31, 2008, until December 31, 2020 (the last released version). Furthermore, we update our dataset with StackExchange Data Explorer. More specifically, we query StackExchange to update our dataset from December 29, 2020 to August 17, 2023. We initially

¹<https://data.stackexchange.com/stackoverflow/>

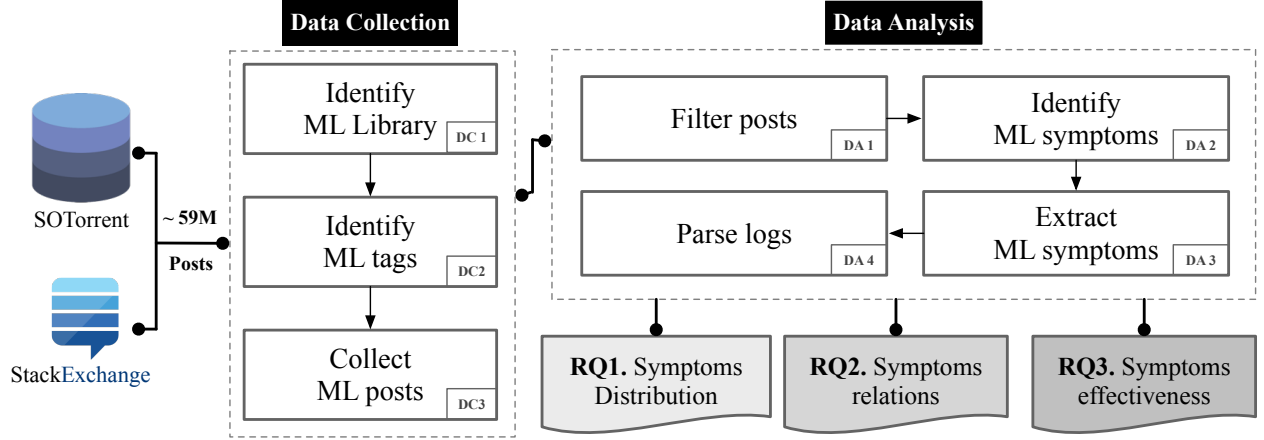


Figure 5.1 An overview of the study design

performed our analysis using the SOTorrent dataset, as SOTorrent provides a convenient way to send queries without limitations. However, as the dataset is not up-to-date, we further update the dataset using the latest Stack Overflow data obtained from StackExchange Data Explore. All of the queries used are available within our replication package [80].

5.4 Data Collection

[DC1] Identify ML-related libraries

To answer our research questions we concentrate on Python libraries that are related to ML applications. Table 5.1 provides summary information about our chosen libraries in terms of their Lines of Code (LOC) written in Python, their age in years, and the number of releases. Excluding *HuggingFace*, a library with a larger codebase (higher LOC) might be more likely to generate questions related to error handling and symptoms. Also, the years (and the number of releases) or maturity of the ML libraries might affect the number of exceptions. We purposefully chose Python projects due to its popularity among data scientists [11, 57] and extensive usage within the ML field [54–56].

Defining ML-related libraries. Our analysis focuses on seven prominent ML libraries offering readily accessible algorithms that enable ML developers to efficiently incorporate ML-based solutions into their applications. These seven well-known ML libraries are *TensorFlow* [13], *PyTorch* [16], *Scikit-learn* [15], *Keras* [14], *NLTK* [17], *HuggingFace* (or *Transformers*) [18], and *Spark ML* (or *MLlib*) [19]. The selection of these ML-related libraries is based on some key considerations: (1) their GitHub popularity metrics (e.g., forks, stars, and releases) [52, 53], (2) their utilization in previous research [11, 54–56], and (3) the imple-

Table 5.1 The volume of the studied ML-related libraries

ML Libraries	LOC ¹	Python LOC	Age (yrs.)	#Release
TensorFlow	3,619,538	737,233 (20.36%)	7	197
Keras	116,522	116,510 (99.98%)	6	52
Scikit-learn	226,590	205,576 (90.72%)	7	35
PyTorch	2,405,376	973,554 (40.47%)	7	46
NLTK	77,555	76,683 (98.87%)	18	76
HuggingFace ²	777,686	770,494 (99.07%)	5	130
Spark ML ³	1,209,170	170,449 (14.09%)	9	73

¹ Line Of Code: we covered only programming language LOCs.

² The Transformers project.

mentation relies on the Python programming language.

[DC2] Identify ML-related tags

To identify ML-related posts, we focus on the question tags provided by SO. These tags aid us in narrowing down our emphasis to questions specifically related to the Python programming language and selected ML libraries. Initially, we define the relevant tags associated with Python ML libraries to guarantee accuracy in our tag selection.

Defining ML-related tag-set. StackExchange helps us find all tag names related to the seven ML libraries on SO. Table 5.2 displays the ML tags used for our investigation. For example, when considering the PyTorch library, we find 12 tag names that include the "pytorch" term within their tags, such as *pytorch-distributions*, and *pytorch-dataloader*.

[DC3] Collect ML-related posts

We use the BigQuery and StackExchange (Data Explorer) services to retrieve posts based on the tag set. To address the research questions, we export all ML-related questions and answers through some queries (available in our replication package [80]). In the end, we extract 131,466 questions, whose statistics are presented in Table 5.4.

Table 5.2 ML library tags

ML Library	Tag Name
TensorFlow	'distributed-tensorflow', 'intel-tensorflow', 'swift-for-tensorflow', 'tensorflow', 'tensorflow-addons', 'tensorflow-agents', 'tensorflow-android', 'tensorflow- c++', 'tensorflow-data-validation', 'tensorflow-datasets', 'tensorflow- estimator', 'tensorflow-extended', 'tensorflow-federated', 'tensorflow- gradient', 'tensorflow-hub', 'tensorflow- io', 'tensorflow-layers', 'tensorflow-lite', 'tensorflow-migration', 'tensorflow- model-analysis', 'tensorflow-model- garden', 'tensorflow-model-optimization', 'tensorflow-probability', 'tensorflow- quantum', 'tensorflow-serving', 'tensorflow-slim', 'tensorflow-ssd', 'tensorflow-transform', 'tensorflow- xla', 'tensorflow.js', 'tensorflow.net', 'tensorflow1.15', 'tensorflow2', 'tensor- flow2.0', 'tensorflow2.x', 'tensorflowjs', 'tensorflowjs-converter', 'tensorflowsharp'
Keras	'auto-keras', 'keras', 'keras-2', 'keras- lambda', 'keras-layer', 'keras-metrics', 'keras-rl', 'keras-tuner', 'keras-vggface', 'tf.keras'
Scikit-learn	'scikit-learn', 'scikit-learn-pipeline'
PyTorch	'gpytorch', 'intel-pytorch', 'pytorch', 'pytorch-dataloader', 'pytorch-datapipe', 'pytorch-distributions', 'pytorch- forecasting', 'pytorch-geometric', 'pytorch-gpu', 'pytorch-ignite', 'pytorch- lightning', 'pytorch3d'
NLTK	'nltk', 'nltk-book', 'nltk-trainer'
HuggingFace	'huggingface', 'huggingface-datasets', 'huggingface-tokenizers', 'huggingface- transformers'
Spark ML	'apache-spark-ml', 'apache-spark-mllib'

5.5 Data Analysis

[DA1] Filter posts

As illustrated in Figure 5.2, we classify the information contained in each SO post into four components: Title, Question, Tags, and Answer/s. Questions and answers can contain various elements, including text, code snippets, etc. To simplify the analysis, we categorize these contents into *Text Blocks* and *Code Blocks*. A *Text Block* includes the textual descriptions provided by the questioner or respondent, while a *Code Block* consists of code snippets, logs, or reports. Our research focuses specifically on questions that include *Code Blocks* containing exception. Although SO often employs the `<pre><code>` HTML tag to indicate *Code Blocks*, there are instances where alternative tags are used. We operate a heuristic approach that uses

multiple regular expressions to identify *Code Blocks* in SO posts to address this variability as much as possible. Through this approach, we automatically extract both *Code Blocks* and *Text Blocks*. Our study's replication package [80] includes the scripts that implement these heuristics.

when I train my network ,It got an error ...

Asked 3 years, 6 months ago Modified 3 years, 6 months ago Viewed 189 times

Title

Text Block { In my train.py

Code Block { `criteon = nn.CrossEntropyLoss()
loss = criteon(binary_output_c1,labels)`

Text Block { both binary_output_c1,labels size are [4, 224,224]...

Code Block { `Traceback (most recent call last):
File "...\\interactiveshell.py", line 3296, in run_code
exec(code_obj, self.user_global_ns, self.user_ns)
File "<...>", line 1, in <module>
runfile(wdir=
File "...\\pydev_umd.py", line 197, in runfile
pydev_imports.execfile(filename, global_vars, local_vars)
File "...\\pydev_execfile.py", line 18, in execfile
file(contents+"\n", file, 'exec'), glob, loc
File "...\\functional.py", line 1881, in nll_loss
out_size, target.size())
ValueError: Expected target size (4, 224), got torch.Size([4, 224, 224])`

Stack Trace **Call Stack** **Exception Type** **Error Message**

Tags python pycharm pytorch

1 Answer Sorted by: Highest score (default)

Text Block { If you are using `nn.CrossEntropyLoss` ...

Code Block { `loss = criteon(torch.cat((-binary_output_c1[:, None, ...], binary_outpu`

Answer/s

Figure 5.2 A truncated example of a SO post. (ID=60798712)

[DA2] Identify ML symptoms

Our research focuses on the symptoms of ML application exceptions. The exception symptoms are Call Stack (CS), Exception Type (Ext), and Error Message (EM), which derive from their pivotal roles in providing comprehensive information about faced errors. These three exception symptoms collectively offer a whole view of the error scenarios, with the Call Stack detailing the sequence of function calls leading to the error, the Exception Type specifying the nature of the error, and the Error Message providing specific details about the encountered issue. By focusing on these symptoms, we aim to capture the essential aspects

that facilitate not only the identification of ML exceptions but also a deeper understanding of the underlying issues. We place abbreviations, meanings, and examples in Table 5.3 to enhance readability.

Table 5.3 ML exception symptoms abbreviations and meanings

Abbreviation	Definition	Example
CS	Call Stack	<pre> File ".../training.py" in _method_wrapper File ".../training.py", in fit File ".../data_adapter.py", in train[...] File ".../nest.py", in map_structure File ".../nest.py", in <listcomp> File ".../data_adapter.py", in _split File ".../ops.py", in convert_to_tensor_v2 File ".../ops.py", in convert_to_tensor File ".../tensor_[...].py" in _default[...] File ".../constant_op.py", in constant File ".../constant_op.py", in _constant_impl File ".../constant_op.py" in convert_... </pre>
ExT	Exception Type	AttributeError
EM	Error Message	'str' object has no attribute 'definition'
EMT	Error Message Template	<*> has no attribute <*>

[DA3] Extract exception symptoms

The indicators (or defined symptoms in the previous step) are typically located within *Code Blocks*. Consequently, we aim to differentiate between *Code Blocks* containing these indicators and other types of *Code Blocks*, such as code snippets or other types of reports.

Call stack mining. In the process of distinguishing the presence of stack traces within *Code Blocks*, a key aspect involves examining the path of libraries and functions, a prominent component of every stack trace. We employ specific regular expressions tailored for diverse operating systems to determine this variation in stack trace structures. These regular expressions serve a dual purpose: 1) dissecting call stacks comprehensively and transforming them into a set of file and function name pairs, and 2) revealing additional elements such as exception types and error messages. Figure 5.3 illustrates the transformation of call stacks into pairs achieved by applying regular expressions.

Error message and exception type mining. Error messages are commonly located in two distinct areas—either at the bottom of stack traces or independently within *Code Blocks*. Conventionally, the exception type and error message are intertwined. Consequently, our regular expressions leverage these patterns and the structure of exception types to pinpoint

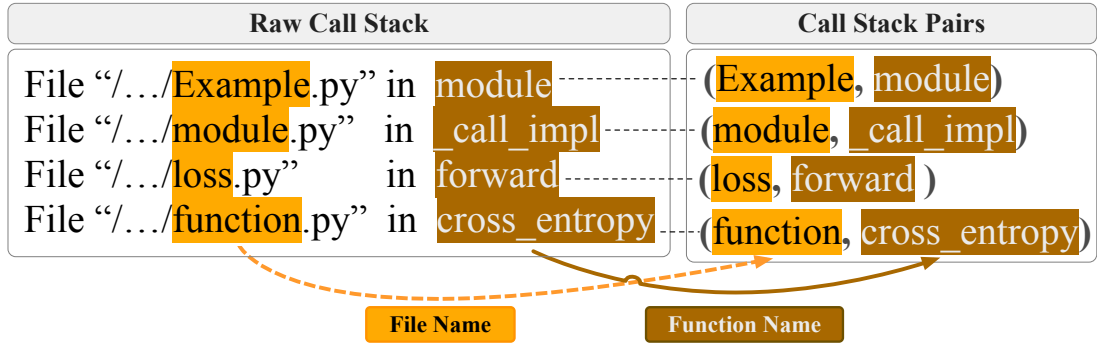


Figure 5.3 A truncated example of call stack transformation. (ID=70716638)

these indicators automatically. Through manual verification, we distinguished that certain keywords, such as "Error", "ERROR", or "Interrupt", are prevalent in the exception type section. Additionally, the error message segment is typically separated by a colon. So, these techniques help us to find exception types and error messages separately.

[DA4] Parse Logs

To find the templates of error messages, we rely on a state-of-the-art log parser to classify error messages accurately. With some modifications, we operate Drain [88], a powerful log template miner. Drain is an online-based log parser capable of continuously transforming logs into organized events. It uses a parse tree with a predefined depth to steer the search for log groups, effectively preventing the creation of excessively deep and unbalanced trees. Drain’s code and our modifications are available in the replication package [80]. Finally, by using Drain, we convert and classify our error messages into error message templates.

5.6 Research Questions and Results

(RQ1) What are the distributions of ML exception symptoms posted on Stack Overflow?

Motivation

When faced with challenges, ML developers frequently turn to Q&A forums, and one of the famous ones is SO. In their posts, developers can determine exceptions or errors that feature stack traces. To answer our research question, we divide a stack trace into exception symptoms. By separating these symptoms we seek to understand them and their role in detecting ML exceptions while gaining insights into the prevailing challenges faced by developers in

the ML community. In addition, we seek to determine whether understanding the symptoms of one exception can provide insights for locating other similar exceptions.

Approach

To answer this RQ, we break down an exception into three essential components: Call Stack (CS), Exception Type (ExT), and Error Message (EM), which we collectively refer to as the "symptoms" of ML exceptions. We then look for these symptoms inside SO questions.

Co-existence of exception symptoms. To understand how symptoms co-exist, we consider diverse combinations of exception symptoms and their spread across various ML-related libraries on SO. The term "combination" refers to the examination of multiple co-occurring symptoms, but also includes the examination of individually occurring symptoms. For the individual symptoms, our attention is directed towards the occurrence of a single symptom on the SO questions, exclusive of other symptoms, such as CS (without ExT and EM). However, we also investigate various co-existence combinations of exception symptoms, such as the existence of ExT and EM (without seeing CS). Additionally, we focus on distributions of questions covering at least one symptom.

Distribution of exception type symptom. We initially classify exception types alongside their respective question IDs to analyze the distribution of exception types compared to their prevalence among SO questions. Subsequently, we arrange them in order of the frequency of occurrence. This methodology allows us to discern the proportion of questions corresponding to each distinct exception type.

Trend of exception symptom. Symptoms generally manifest within the *Code Block* sections or are integrated into stack traces. However, our results reveal that in some cases, ML questioners on SO sometimes opt to put exception symptoms (exception types and error messages) within distinct *Code Blocks*. The evolving trend of this manner across various years is depicted in Figure 5.4. In essence, our investigation delves into identifying symptoms present in diverse formats of *Code Blocks*, containing those with and without stack traces (or call stacks).

Parse Logs. We employ a log parser to organize and classify numerous error messages. We perform log parsing for several reasons: Firstly, it allows automated categorization, ensuring an efficient and consistent method of handling various kinds of error messages. A manual technique can be time and energy-consuming and prone to inconsistencies. Additionally, a log parser helps extract valuable insights from the error messages to identify their patterns. With some modifications, we operate Drain [88], a powerful log template miner. Drain is an online-

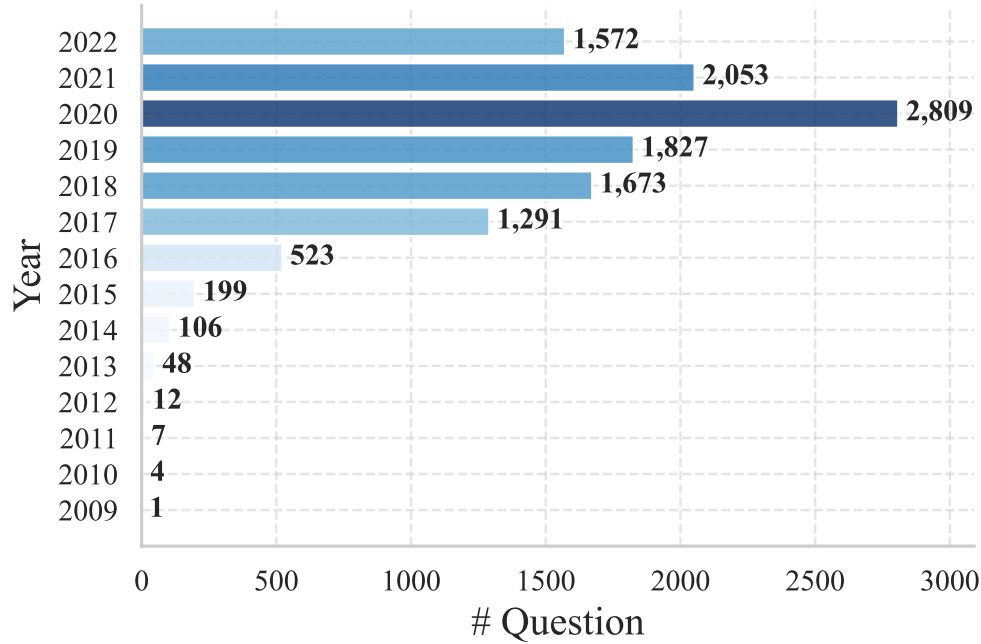


Figure 5.4 The trend of questions with at least one Error Message and without any Call Stack

based log parser capable of continuously transforming logs into organized events. It utilizes a parse tree with a predefined depth to steer the search for log groups, effectively preventing the creation of intense and unbalanced trees. The Drain's code and our modifications are available in the replication package [80]. We establish a ground truth using 379 samples to validate the approach, aiming for a 95% confidence level while maintaining a 5% confidence level to ensure its reliability.

Results

A large portion (24.31% in average) of ML-related questions contain at least one exception symptom. Table 5.4 provides a comprehensive overview of the distribution of various symptom combinations across different ML libraries on SO. Each row corresponds to a specific ML library, and the columns detail the number of questions (#Ques.) and the occurrences of different symptom combinations, including Call Stacks (CS), Exception Types (ExT), and Error Messages (EM). Notably, the last column, labeled "At least one Symp.", is particularly interesting. It reveals the percentage of ML-related questions within each library that feature at least one of the analyzed exception symptoms. On average, 24.31% of ML-related questions (spans from 16.50% for *NLTK* to 31.95% for *HuggingFace*) demonstrate at least one exception symptom. This statistic underscores the prevalence and importance of

encountering and addressing exception-related challenges within the ML community on SO.

Only half (33.65% to 57.19%) of the questions with exception symptoms carry all three symptoms, while (32.35% to 56.81%) of the exception questions contain only the exception type and error message symptoms. Examining the distribution in the last column of Table 5.4, denoted as *At least one symptom*, shows the prevalence of exception questions (and their percentages compared to the total number of questions) within each ML library. Remarkably, only a fraction of exception questions, ranging from 33.65% to 57.19%, enclose all three symptoms—Call Stack, Exception Type, and Error Message. However, a substantial portion, ranging from 32.35% to 56.81%, involves exception questions with only the Exception Type and Error Message symptoms. The second to last column reveals the specific count of questions where all three symptoms coexist. On average, across ML libraries, only 47.51% of exception questions exhibit all three symptoms. Similarly, the presence of both Exception Type and Error Message symptoms is observed in an average of 42.54% of exception questions.

Table 5.4 The distribution of symptom combinations

ML Library	#Ques.	At least one Symp.		Symptom Combinations						
				#CS ¹	#ExT ¹	#EM ¹	# CS&EM	# CS&ExT	# ExT&EM	# CS&ExT&EM
TensorFlow	55,257	14,291	(25.86%)	305	21	0	0	79	6,314 (44.18%)	7,457 (52.17%)
Keras	30,620	7,895	(25.78%)	148	13	0	0	49	3,723 (47.15%)	3,924 (49.70%)
Scikit-learn	23,898	5,033	(21.06%)	99	7	0	0	79	2,304 (45.77%)	2,526 (50.18%)
PyTorch	13,481	3,177	(23.57%)	92	2	0	0	31	1,205 (37.92%)	1,817 (57.19%)
NLTK	6,426	1,060	(16.50%)	45	1	0	0	68	396 (37.35%)	523 (49.33%)
HuggingFace	1,618	517	(31.95%)	20	3	0	0	3	174 (33.65%)	314 (60.73%)
Spark ML	166	44	(26.51%)	1	0	0	0	0	25 (56.81%)	18 (40.90%)

¹ Abbreviation of symptoms: Call Stack (CS), Exception Type (ExT), and Error Message (EM)

- #CS, #ExT, #EM: These columns individually count the number of questions that include Call Stack (CS),

Exception Type (ExT), and Error Message (EM) information, respectively.

- #CS & EM, #CS & ExT, #ExT & EM, #CS & ExT & EM: These columns

indicate the number of questions that contain combinations of the symptoms.

- **At least one Symptom:** It calculates the number and percentage of questions within each ML library category containing at least one symptom (CS, ExT, or EM). This percentage represents the proportion of questions where some form of exception-related information is provided.

A small portion of exception types covers the majority of the exception questions. The cumulative Exception type percentage, as illustrated in Figure 5.5, reveals an interesting pattern in the distribution of exception questions. Specifically, the top 20% of all exception types account for more than 90% of the total exception questions. This implies that a relatively small subset of exception types are related to most questions. Remarkably, excluding *Spark ML* and *HuggingFace*, a mere 10% of exception types encloses over 80% of all exception questions. This observation underscores the significance of focusing on specific exception types that recurrently emerge when addressing programming challenges, providing valuable insights for targeted support and documentation efforts.

A small portion of error message templates cover the majority of the exception

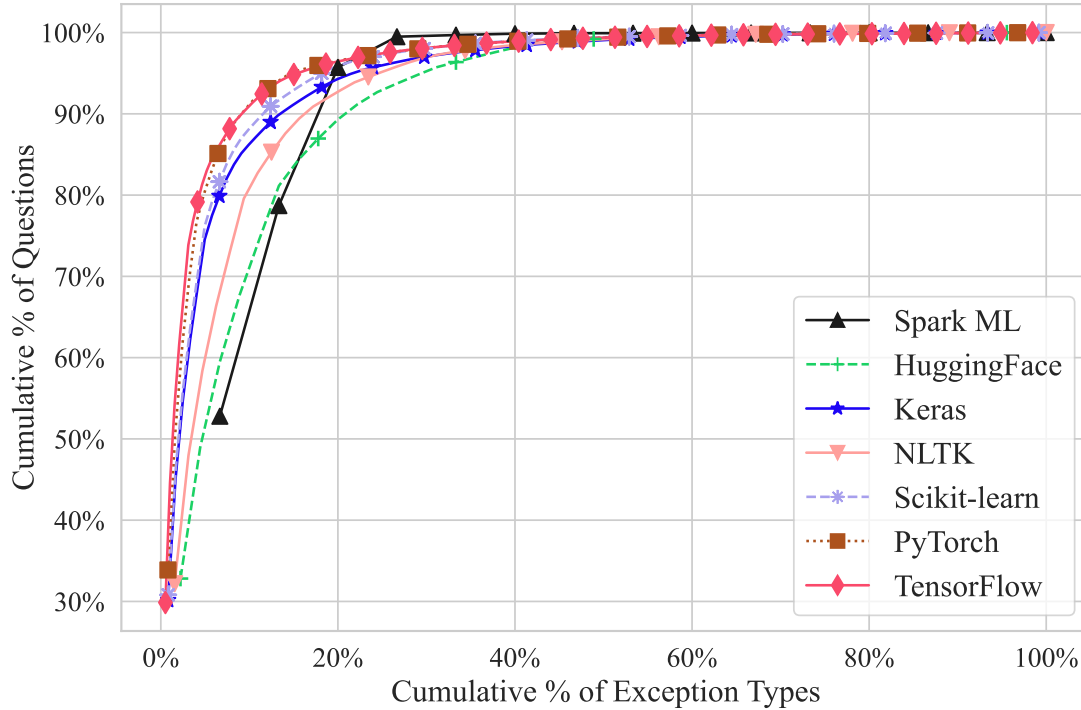


Figure 5.5 The cumulative percentage of ML-related questions vs. the percentage of Exception types

questions. Similar to the insights drawn from Figure 5.5, Figure 5.6 offers a comprehensive view of the cumulative distribution of Error Message Templates across varying proportions of exception questions. The data highlights that a mere 10% of Error Message Templates covers over 75% of all exception questions across different ML-related libraries. This observation serves as an important indicator, shedding light on the specific challenges or complexities ML developers face during their implementations. The prevalence of a focused set of EMTs implies that directing attention to these templates can greatly improve the effectiveness of troubleshooting and support initiatives.

Table 5.5 illustrates the top 20 most frequently occurring Exception Types (Ext) in the question *Code Blocks*. These Exception Types are important indicators of errors and exceptions within ML-based applications. The table provides a clear overview of the frequency with which each Exception Type appears, with "ValueError" leading the list at 9,018 occurrences, followed by "TypeError", and "AttributeError" with 4,308 and 2,778 presence, respectively. This information emphasizes the common issue and challenge types developers face in the ML domain, helping to prioritize areas of focus for improving the reliability of ML applications or applying diagnosing strategies. Although a substantial majority of top ExTs are part

Table 5.5 Top 20 most frequent Exception Types

Exception Type	Count
ValueError	9,018
TypeError	4,308
AttributeError	2,778
ImportError	2,220
InvalidArgumentError	1,882
RuntimeError	1,727
ERROR	1,227
ModuleNotFoundError	826
IndexError	588
KeyError	489
OSError	383
NotFoundError	325
NameError	281
AssertionError	269
Exception	256
FailedPreconditionError	238
InternalError	188
NotImplementedError	172
SyntaxError	171
MemoryError	160

of Python's built-in exception hierarchy, some appear to have been introduced outside the standard Python framework. The definitions of Python's standard ExTs can be accessible through this link². However, upon closer examination, we discovered that the *"ERROR"* ExT is associated with the installation of ML libraries, like 64066644, created based on interactions with other programming languages (like C/C++) or custom error-handling methods, such as 62058265. It's evident that the ExTs which are distinct from Python's built-in exceptions, such as *"ERROR"*, *"NotFoundError"*, *"InvalidArgumentError"*, *"InternalError"*, and *"Failed-PreconditionError"*, except one of them, are not self-explanatory regarding the primary cause of the associated exceptions. To address this problem, we can implement a custom exception handler or an error mapping tool that establishes connections among different project pieces that utilize diverse exception types.

²<https://docs.python.org/3/library/exceptions.html>

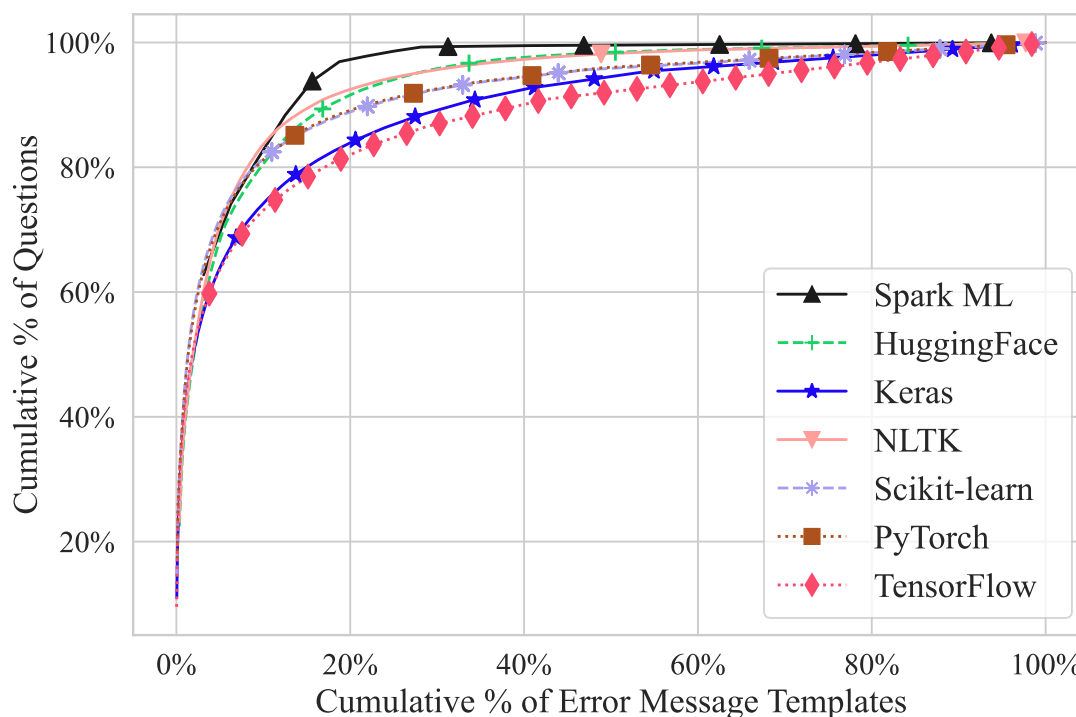


Figure 5.6 The cumulative percentage of ML-related questions vs. the percentage of Error Message Templates

A significant portion of exception-related questions contain only the call stack symptom, whereas a much smaller portion of exception-related questions contain only the exception type or only the error message symptoms. Analyzing the data in the fourth column of Table 5.4 (#ExT), we delve into questions lacking both a Call Stack and an Error Message. This exploration reveals that at least two prominent Exception Types lack any explanatory information or associated Error Messages. These two Exception Types are: ***AssertionError*** and ***MemoryError***. The "*AssertionError*" type can be relevant to Python and Java programming languages. "*AssertionError*" is raised when an assert statement fails³, and we note that they tend to provide a general indication without showing the root cause of the error. A similar situation is observed with "*MemoryError*".

³<https://docs.python.org/3/library/exceptions.html#AssertionError>

Summary of RQ1

About half of the exception-related ML questions (i.e., containing at least one exception symptom) contain all three exception symptoms (call stacks, exception types, and error messages). Most of the other exception-related ML questions either contain a call stack or a combination of an exception type and an error message. Developers' various ways of posting exception symptoms motivate us to investigate how the symptoms impact the ambiguity (RQ2) and community support (RQ3) of exception-related ML questions.

(RQ2) How ambiguous are the different symptoms of ML exceptions?

Motivation

RQ1 reveals that ML developers post different types of ML exception symptoms. Nevertheless, a crucial aspect that remains unclear is how these exception symptoms, when not presented concurrently, can unambiguously identify ML exceptions. In the context of this research question, we investigate the relations among exception symptoms, with a specific emphasis on exploring the potential ambiguities that arise when only a subset of symptoms is provided.

Approach

Relation between exception types and error message templates. Table 5.6 is divided into two sections: one detailing the distribution of Exception Types for each Error Message Template and the other showing the distribution of Error Message Templates per Exception Type. To clarify our procedure for obtaining the numbers on Exception Types (#ExT) and the most frequent Exception Types columns, a consistent approach is applied for both sections. Initially, we identify all Exception Types corresponding to each Error Message Template, capturing their frequencies and highlighting the most common ones. Subsequently, we compute the Median (Med.) and Interquartile Range (IQR) (i.e., $Q_3 - Q_1$) for each distribution.

Calculating pair-wise similarity between two call stacks. To compute the pair-wise similarity between Call Stacks, we employ equation 5.1, where C_i represents the set of pairs for each call stack, as illustrated in Figure 5.3.

$$Sim(C_1, C_2) = \frac{C_1 \cap C_2}{\min(C_1, C_2)} \quad (5.1)$$

Equation 5.1 quantifies the pair-wise similarity by dividing the number of common call stack pairs between two Call Stacks by the length of the shorter Call Stack. It is crucial to note that in converting a raw call stack into a call stack pair, we disregard the order of pairs, retaining only the unique pairs of file and function names. Crucially, the similarity value is a variable that falls within the spectrum of 0 to 1, reflecting the level of resemblance between the two Call Stacks. This normalized scale measures the extent of similarity and dissimilarity, providing a valuable metric for comparing various Call Stacks in our study.

Analyzing the Relationship between Call Stacks and Exception Types/Error Message Templates. Table 5.7 intricately presents the distribution of Call Stacks per Exception Type and Error Message Template, with a clear separation into two sections. While the methodology for generating these numbers is nearly identical for both segments, we focus on the process using the first part for clarity. For each Exception Type, a list of Call Stacks is collected. As a reminder, each Call Stack comprises a set of pairs outlined in Figure 5.3. To quantify the similarity between every pair of Call Stacks within the list associated with each Exception Type, we employ the pair-wise similarity method (i.e., the combination of two from the Call Stack list without considering the order and replacement). This process yields a list of numerical values from which we extract a range. Then, we report the minimum, median, and maximum pair-wise similarity numbers of them. Finally, we calculate the Median (Med.) and Interquartile Range (IQR) for each respective part of this range.

Results

An error message template is associated with a median of 2 to 3 exception types, whereas an exception type is associated with a median of 7 to 33 error message templates; thus, using error message templates or exception types individually cannot unambiguously identify exception-related questions. Table 5.6 sheds light on the intricate connection between Exception Types and Error Message Templates across various ML libraries. The table is structured into two parts: one outlining the distribution of Exception Types corresponding to each Error Message Template and the other presenting the distribution of Error Message Templates associated with each Exception Type. Upon delving into the first part, we observe that, for each Error Message Template, there is a median of two distinct Exception Types across all ML libraries, except for *NLTK* and *HuggingFace*, where the median is three. Notably, the Interquartile Range for *NLTK*, *HuggingFace*, and *Spark ML* surpasses that of other libraries (11, 17, and 36.2, respectively), indicating a more diverse set of Exception Types for each Error Message Template in these libraries, as opposed to the typical two or three found in others. An investigation of the most frequent Exception

Type column in this part reveals a symmetric and evenly spread distribution of frequencies within the central region of the median, with exceptions noted for *NLTK*, *HuggingFace*, and *Spark ML*. On the other hand, the second part of the table illuminates a notable finding: each Exception Type demonstrates a diverse association with a median range spanning from 7 to 33 distinct Error Message Templates across a spectrum of ML libraries. Additionally, when examining the most frequent Error Message Templates per Exception Type, the range extends from 2 to 12. This variation underscores the rich diversity and complexity of how Exception Types are linked to a multitude of Error Message Templates within different ML libraries.

Table 5.6 Distribution of Exception Types per Error Message Template and vice versa

ML Library	Dist. of Exception Type per Error Message Template				Dist. of Error Message Template per Exception Type			
	# ExT		Most Freq. ExT		# EMT		Most Freq. EMT	
	Med.	IQR	Med.	IQR	Med.	IQR	Med.	IQR
TensorFlow	2.0	3.0	2.0	2.0	7.0	30.0	3.0	9.0
Keras	2.0	5.0	2.0	4.0	21.0	73.0	6.0	22.0
Scikit-learn	2.0	4.0	2.0	3.0	14.5	61.0	4.0	16.0
PyTorch	2.0	4.0	2.0	3.0	12.0	41.0	3.0	12.0
NLTK	3.0	11.0	3.0	8.0	29.0	166.0	8.5	46.5
HuggingFace	3.0	17.0	2.0	13.0	33.0	416.0	12.0	70.0
Spark ML	2.0	36.2	2.0	12.8	8.0	352.5	2.0	100.5

An exception type or an error message template can be associated with completely different call stacks. Table 5.7 depicts the distribution of Call Stacks for each Exception Type and Error Message Template in two distinct scenarios. In both cases, the median similarity of call stacks is consistently zero, except for *TensorFlow*, which exhibits a median of two. This observation highlights the considerable diversity in associations between Exception Types/Error Message Templates and their corresponding Call Stacks. The low median pair-wise similarity suggests that Exception Types and Error Message Templates might not serve as robust representatives for their Call Stacks, as there is often minimal similarity among the associated Call Stacks.

Lack of attribute is the most common exception. Table 5.8 lists the top 30 most frequent Error Message Types (EMTs) and their respective frequencies. The table reveals that the most frequent error message belongs to the "<*><*>has no attribute <*>" template with 2,487 occurrences, displaying problems related to missing attributes in objects. The second most frequent EMT is "No module <*><*>" with 1,237 instances, suggesting

Table 5.7 Distribution of Call Stacks for each Exception Type and Error Message Template

ML Library	Dist. of Call Stack per Exception Type						Dist. of Call Stack per Error Message Template					
	Pair-wise Similarity ¹						Pair-wise Similarity ¹					
	Min.		Med.		Max.		Min.		Med.		Max.	
	Med.	IQR	Med.	IQR	Med.	IQR	Med.	IQR	Med.	IQR	Med.	IQR
TensorFlow	0.0	0.0	0.0	0.2	1.0	0.0	0.0	1.0	0.2	0.8	1.0	0.5
Keras	0.0	0.0	0.0	0.2	1.0	0.0	0.0	1.0	0.0	0.6	1.0	0.4
Scikit-learn	0.0	0.0	0.0	0.2	1.0	0.0	0.0	1.0	0.0	0.5	1.0	0.5
PyTorch	0.0	0.0	0.0	0.2	1.0	0.0	0.0	1.0	0.0	0.7	1.0	0.7
NLTK	0.0	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.3	1.0	0.0
HuggingFace	0.0	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.5	1.0	0.0
Spark ML	0.0	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.3	1.0	0.0

¹ Pair-wise Similarity within Exception Types/Error Message Templates results in a numerical range, encompassing minimum, median, and maximum values.

problems in locating required Python modules. This problem comes from two common clues: first, a package or module is missing, or we have a version conflict. Other common EMTs include errors related to attribute and module imports, array shapes, unexpected keyword arguments, and inconsistent data types, among others, reflecting the common challenges faced by ML developers and debuggers.

Following our exploration, within the $\langle * \rangle$ template, the asterisk in this template denotes either numeric values or variable names, commonly associated with errors arising from attempts to access a key in a Python dictionary. The prevalent exception type in this context is "KeyError," exemplified by instances such as 59246768 or 58085041, as indicated in our analysis. Furthermore, the scenario becomes more challenging for the *EMPTY* template, marked as the 10th row with 270 occurrences. This particular template signifies a lack of information available to reveal the root cause of the error, such as 69885845 or 58837809. Similar to the exception types outlined in Table 5.8, this kind of error message and exception type needs additional contextual information to disclose the underlying cause for developers. In other words, the presence of just a singular name or number or even nothing does not convey the comprehensive and short details required to determine the root cause of the error. This highlights a need for more informative error messages or documentation to aid developers in effectively addressing these issues.

Table 5.8 Top 30 most frequent Error Message Templates

Error Message Template	Count
<*><*>has no attribute <*>	2,487
No module <*><*>	1,237
in <*>code:	640
Error when checking <*>expected <*>to have <*><*>but got array with shape <*>	551
<*>object is not <*>	437
<*>	388
cannot import name <*>	331
DLL load failed: <*>specified <*>could not be <*>	289
<*>got an unexpected keyword argument <*>	279
{EMPTY}	270
name <*>is not <*>	261
setting an array element with a <*>	243
cannot import name <*>from <*><*>	224
<*>missing <*>required positional argument: <*>	211
No <*><*>found for <*>	198
list index out of <*>	180
Exception encountered when calling layer <*><*><*>	166
Found input variables with inconsistent numbers of samples: <*>	156
Graph execution error:	149
You must feed a value for placeholder tensor <*>with 'dtype <*>and shape <*>	146
<*>to use <*>value <*>	145
Incompatible shapes: <*>vs. <*>	141
<*>root error(s) found.	137
<*>cannot open shared object file: No such file <*><*>	137
invalid <*>	133
could not convert string to <*><*>	132
unsupported operand type(s) for <*><*>and <*>	119
Expected 2D array, got <*>array <*>	116
<*>takes <*>positional <*>but <*>were given	110
You must feed a value for placeholder tensor <*>with dtype <*>	108

Summary of RQ2

Posting error message templates or exception types individually cannot unambiguously identify exception-related questions. Besides, an exception type or an error message template can be associated with completely different call stacks. Thus, we suggest developers post the complete symptoms of ML exceptions to reduce the ambiguity of their posts. Post searchers should also leverage as many symptoms as possible to maximize their chance of uniquely identifying similar questions related to ML exceptions.

(RQ3) How do the exception symptoms impact community support?

Motivation

The earlier research questions have established that ML developers share various types of ML exceptions, and these symptoms, when presented individually, lack the clarity to identify ML exceptions distinctly. This research question tries to understand the impact of different exception symptoms on community support. We hypothesize that presenting symptoms more unambiguously increases the likelihood of receiving timely support from the community.

Approach

Metrics for assessing community support. Below, we describe the metrics that we study to determine the level of community support among distinct combinations of symptoms on SO.

- **View Count:** This metric serves as an indicator of popularity. SO tracks the number of views per page load⁴. It represents the total number of views a particular question has received.
- **First Answer Duration:** a metric measures the duration (hours) it takes for the first response to be posted.
- **Accepted Answer Duration:** This metric quantifies the time span, in hours, between the publication of a question and the moment it garners an accepted answer (chosen by the questioner as the solution to their question).
- **Answer Rate:** The answer rate refers to the percentage of questions that receive the accepted answer. It provides insights into how quickly and effectively the community responds to posted questions.

Measuring the community support for different symptom combinations. We examine the final three symptom combinations outlined in Table 5.4, specifically *# ExT and EM*, *# CS, ExT and EM*, and *At least one Symptom*, to assess their community support across ML libraries. Analyzing community support metrics for questions with and without exception/s provides insights into the impact of exception symptoms on the level of engagement and assistance within the community.

⁴<https://meta.stackexchange.com/questions/36728/how-are-the-number-of-views-in-a-question-calculated>

Calculating the cosine similarity between post’s title and error message. The cosine similarity approach, equation 5.2 helps us compute the similarity between post’s titles ($\mathbf{T}$) and error messages ($\mathbf{E}$). The formula serves as a quantitative measure to assess the similarity between the vector representations.

$$\text{Cosine Similarity}(\mathbf{T}, \mathbf{E}) = \frac{\mathbf{T} \cdot \mathbf{E}}{\|\mathbf{T}\| \times \|\mathbf{E}\|} \quad (5.2)$$

The resulting cosine similarity value ranges from -1 to 1, where 1 signifies perfect similarity, 0 indicates no similarity, and -1 represents perfect dissimilarity. In other words, a higher cosine similarity suggests a greater semantic resemblance between the title and Error Message. Utilizing the numerical outputs derived from the cosine similarity formula and employing a threshold ($\theta = 0.3$), we classify questions into two distinct groups: those deemed similar and those categorized as not similar (dissimilar). The selection of the threshold (θ) is determined through a manual analysis of the results from both groups, ensuring a judicious choice for an appropriate threshold that effectively separates questions into these distinct categories.

Results

Questions with only the call stack symptoms are least likely to receive a timely answer. Figure 5.7 illustrates the community support levels for questions exhibiting various combinations of exception symptoms, such as those with At least one Sympt., CS&ExT&EM, ExT&EM, and CS. The metrics considered for community support encompass *First Answer Duration*, *Accepted Answer Duration*, and *View Count* across seven ML-related libraries. The findings suggest that questions with Call Stacks are more challenging, or developers are less likely to respond to lengthy questions involving a Call Stack. Furthermore, Table 5.9 presents the percentage of questions obtaining an accepted answer based on different combinations of exception symptoms. Once more, the table highlights that, in nearly all instances, questions featuring a Call Stack exhibit a lower acceptance rate than other symptom combinations.

Adding exception types and error messages without call stacks increases the chance for a question to receive a timely accepted answer. As depicted in Figure 5.7, questions with Exception Type and Error Message experience shorter waiting times for the accepted answer, ranging from 0.1 to 14.3 hours, in contrast to situations where questions contain Call Stacks. This may be due to exception types and error messages being more informative (or more similar to natural language) than Call Stacks.

Surprisingly, questions with all three exception symptoms are less likely to receive a timely answer than questions with only exception types and error messages.

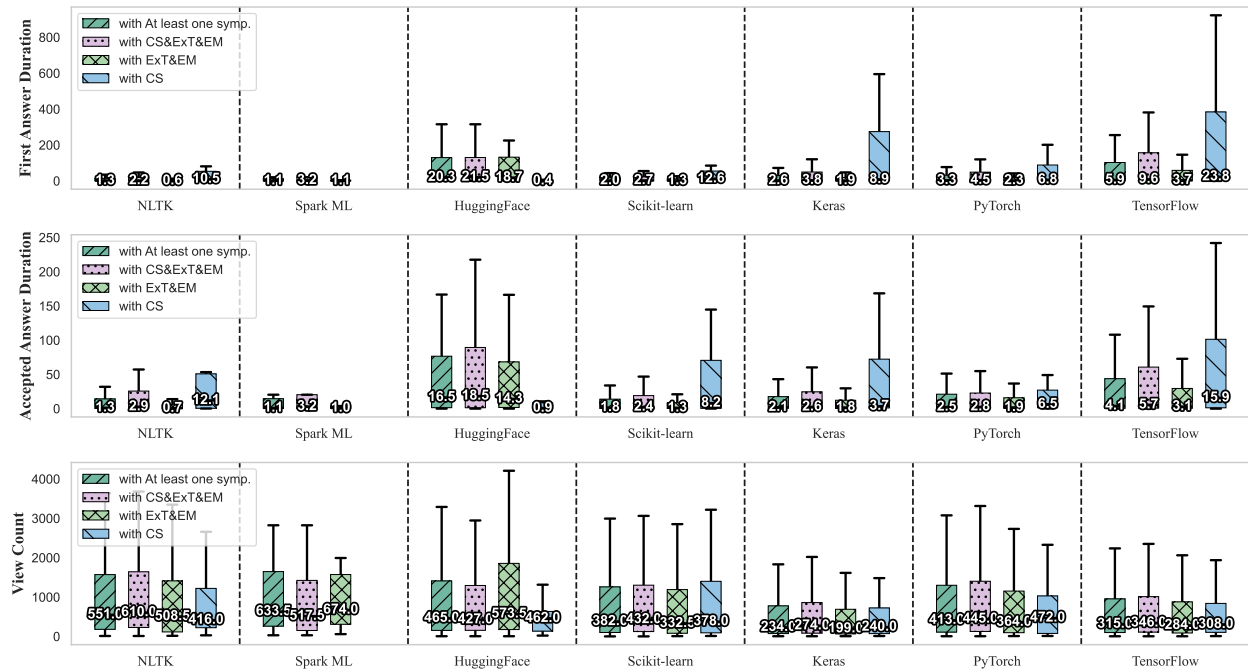


Figure 5.7 The level of community support for questions with different combinations of exception symptoms.

Table 5.9 The percentage of questions receiving an accepted answer for different exception symptom combinations.

ML Library	Symptom Combinations			
	At least one Symp.	# CS&ExT&EM	# ExT&EM	#CS
TensorFlow	33.36%	30.29%	37.3%	26.24%
Keras	37.68%	34.36%	41.48%	31.91%
Scikit-learn	42.28%	39.73%	46.07%	22.68%
PyTorch	33.98%	31.42%	38.4%	32.18%
NLTK	42.45%	40.73%	48.23%	31.11%
Hugging Face	34.43%	32.48%	39.08%	30.0%
Spark ML	43.18%	50.0%	40.0%	-

Placing the error message information in the post title may not increase the likelihood of receiving a timely answer from the community. This result may indicate that developers may not find Call Stacks useful when they try to answer a question or that a question with a call stack indicates a complex one.

Including the error message in the post's title leads to a quicker response. Fig-

ure 5.8 compares a situation in which questioners choose post's title words among Error Message terms. Evidently, posts that have similarities attract more attention, and their view count is higher across all ML libraries. When disregarding *Spark ML*, the differences in accepted answer duration for posts have similarities, and dissimilarities are negligible amongst ML libraries. However, *TensorFlow* stands out with a more favorable status.

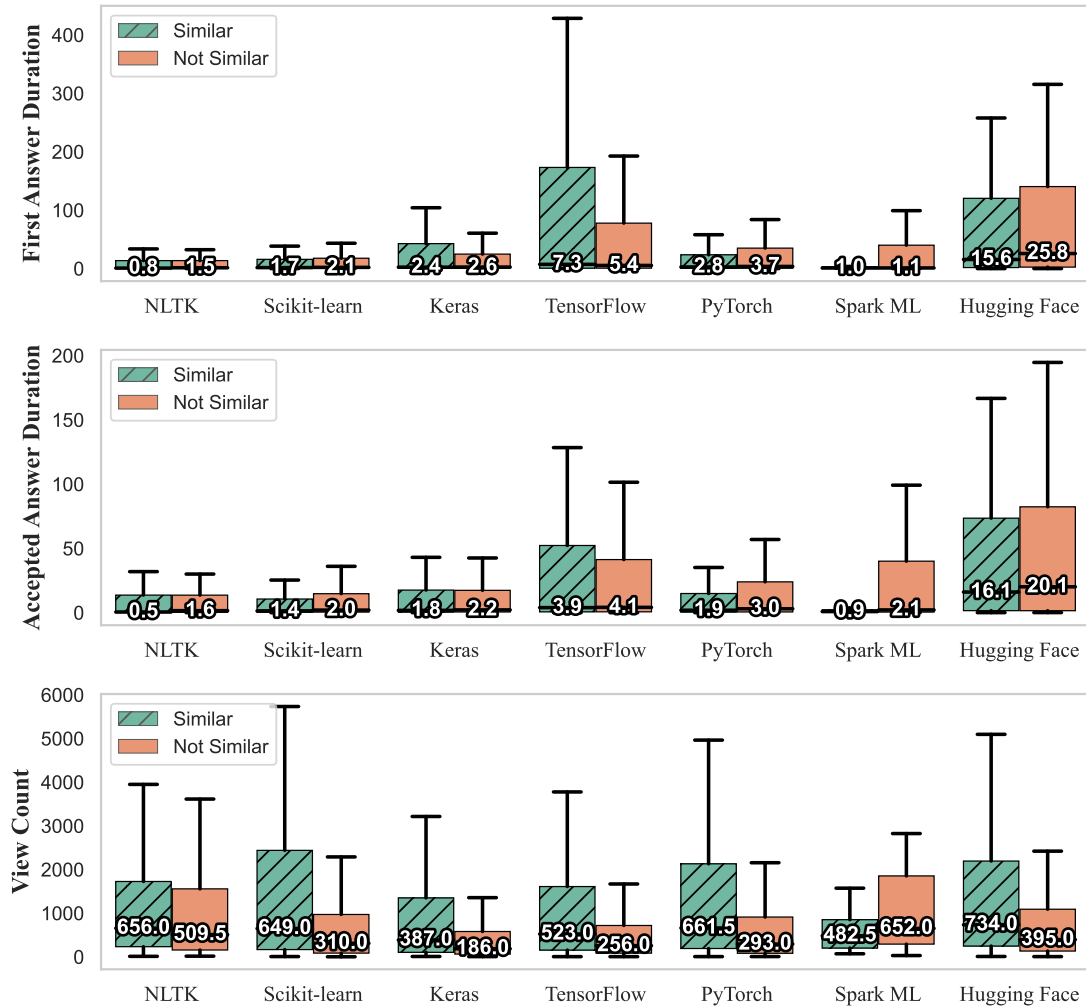


Figure 5.8 The status of community support among questions with and without similarity between Error Message and the title of exception questions. Numbers on the box plots illustrate their medians.

Summary of RQ3

Surprisingly, ML questions with all three exception symptoms are not the most likely to receive timely community support. In fact, questions with only the call stack symptoms are least likely to receive a timely answer, and adding exception types and error messages to the call stacks can increase the likelihood and speed of receiving an accepted answer.

5.7 Threats to Validity

External validity threats. Our choice of SO as the main data source is grounded in its widespread popularity, significance in the developer community [89], and existing works. However, this choice may hamper the generalizability of our results. To improve this generalizability, future research efforts can consider incorporating additional sources beyond SO, such as *GitHub*. Additionally, our current study centralizes posts associated with seven specific ML libraries and Python programming language, which, while providing valuable insights, may not generalize to other libraries. Expanding our analysis to include other programming languages and a more diverse set of ML libraries can further support the external validity of our research. However, we believe that due to the popularity of our chosen data, our results can affect a broad range of developers.

Two distinct approaches exist for filtering SO posts, either with or without discrimination. Some studies, such as [78] and [79], opted to select posts based on popularity. In contrast, our study intentionally refrained from applying any specific criteria to filter ML posts, leading to enhanced insights into the landscape of ML-related posts on SO. Overall, an intriguing avenue for future exploration can involve filtering and comparing ML exception symptoms based on the versions and releases of ML libraries.

Internal validity threats: Developers have the flexibility to truncate exceptions before sharing them on SO, and our analysis revealed instances where the structure of exceptions was altered during posting. In order to address this potential threat, we decompose exceptions into components and conduct separate statistical analyses on each component (refer to Table 5.4) to identify irregular cases. Additionally, while the regular expressions employed for exception identification may not cover all scenarios, we ensure accuracy through manual verification checks.

Construct validity threats. The alignment between SO and SOTorrent, which is dated as of December 31, 2020, remains ambiguous. Nonetheless, several previous studies [90–93] leverage and derive sensible insights from SOTorrent. Additionally, due to the considerable

amount of input data, it is impractical to individually verify all results. Despite conducting manual reviews, the potential exists for overlooking rare cases that are challenging to identify. Therefore, to enable the verification of our results, we offer the source code and input data associated with this study in our replication package [80].

CHAPTER 6 CONCLUSION

To understand the characteristics of ML stack traces and uncover their patterns and challenges, we perform a large-scale quantitative and qualitative study of the Stack Overflow posts that contain Python stack traces related to seven popular ML libraries. Using this data, we manually identify stack trace pattern types and their categories. The first study reveals that ML stack traces on Stack Overflow are common and tend to garner more comments and views than questions without stack traces. However, they are less likely to have accepted answers. We use a common pattern mining algorithm to find that recurrent patterns are common in the stack traces of ML questions and that these patterns can be shared across multiple ML libraries. Finally, we manually classify 995 stack traces into five high-level categories and their challenges and find that misunderstandings in ML API usages, data formats, and language constructs are prevalent causes of errors.

Our results provide insights for future efforts from researchers, ML library developers, and technical forum moderators to better support ML developers to write error-free ML code. For example, technical forum moderators could pay particular attention to prioritizing ML questions with stack traces as they are less likely to receive timely accepted answers. Researchers and ML library developers could prioritize efforts to help developers identify misuses of ML APIs (to tackle model-related exception patterns), mismatches in data formats (for data-related exception patterns), or potential data/resource contentions (for multi-process related patterns).

The second study demonstrates the symptoms of exceptions and assesses their impact on community support in resolving failures. Our focus extends beyond call stacks to contain additional aspects of exceptions, specifically the exception type and error message. This study shows the distribution of ML exception symptoms on Stack Overflow, highlighting that approximately half of the questions involve all three symptoms, while others feature specific combinations. The exploration of symptom ambiguity indicates that individual error message templates or exception types do not definitively identify ML exception-related questions. Notably, questions solely featuring call stack symptoms are less likely to receive prompt community support. However, incorporating exception types and error messages alongside call stacks enhances the likelihood and speed of obtaining accepted answers. We hope our results can help improve how exceptions are used on forums such as Stack Overflow.

In future work, we plan to create a tool to help developers quickly find solutions to their ML-related problems on Stack Overflow based on the symptoms of ML exceptions. It is our

hope that our results can also help improve how stack traces are used on forums such as Stack Overflow.

REFERENCES

- [1] M. I. Jordan and T. M. Mitchell, “Machine learning: Trends, perspectives, and prospects,” *Science*, vol. 349, no. 6245, pp. 255–260, 2015.
- [2] NewVantage Partners, “The quest to achieve data-driven leadership: A progress report on the state of corporate data initiatives,” 2022. [Online]. Available: https://c6abb8db-514c-4f5b-b5a1-fc710f1e464e.filesusr.com/ugd/e5361a_2f859f3457f24cff9b2f8a2bf54f82b7.pdf
- [3] Deloitte, “Annual report 2020/21,” https://www2.deloitte.com/content/dam/Deloitte/dk/Documents/about-deloitte/Impact_Report_20_21_web.pdf.
- [4] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, “A comprehensive study on deep learning bug characteristics,” in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2019, pp. 510–520.
- [5] T. Zhang, C. Gao, L. Ma, M. Lyu, and M. Kim, “An empirical study of common challenges in developing deep learning applications,” in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2019, pp. 104–115.
- [6] M. Alshangiti, H. Sapkota, P. K. Murukannaiah, X. Liu, and Q. Yu, “Why is developing machine learning applications challenging? a study on stack overflow posts,” in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2019, pp. 1–11.
- [7] A. Hamidi, G. Antoniol, F. Khomh, M. Di Penta, and M. Hamidi, “Towards understanding developers’ machine-learning challenges: A multi-language study on stack overflow,” in *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2021, pp. 58–69.
- [8] A. A. Bangash, H. Sahar, S. Chowdhury, A. W. Wong, A. Hindle, and K. Ali, “What do developers know about machine learning: a study of ml discussions on stackoverflow,” in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 260–264.

- [9] M. J. Islam, H. A. Nguyen, R. Pan, and H. Rajan, “What do developers ask about ml libraries? a large-scale study using stack overflow,” *arXiv preprint arXiv:1906.11940*, 2019.
- [10] P. Chandrasekar. (2020) Scripting the future of stack overflow. [Online]. Available: <https://stackoverflow.blog/2020/01/21/scripting-the-future-of-stack-2020-plans-vision/>
- [11] “Stack overflow developer survey 2021.” [Online]. Available: <https://insights.stackoverflow.com/survey/2021#overview>
- [12] X. Sun, T. Zhou, G. Li, J. Hu, H. Yang, and B. Li, “An empirical study on real bugs for machine learning programs,” in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2017, pp. 348–357.
- [13] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, “{TensorFlow}: A system for {Large-Scale} machine learning,” in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 265–283.
- [14] F. Chollet *et al.* (2015) Keras. [Online]. Available: <https://github.com/fchollet/keras>
- [15] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, “Scikit-learn: Machine learning in python,” *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.
- [16] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimeshein, L. Antiga *et al.*, “Pytorch: An imperative style, high-performance deep learning library,” *Advances in neural information processing systems*, vol. 32, 2019.
- [17] E. Loper and S. Bird, “Nltk: The natural language toolkit,” *arXiv preprint cs/0205028*, 2002.
- [18] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz *et al.*, “Huggingface’s transformers: State-of-the-art natural language processing,” *arXiv preprint arXiv:1910.03771*, 2019.
- [19] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar, “Mllib: Machine learning in apache spark,” *J. Mach. Learn. Res.*, vol. 17, no. 1, p. 1235–1241, jan 2016.

- [20] H. Atwi, B. Lin, N. Tsantalis, Y. Kashiwa, Y. Kamei, N. Ubayashi, G. Bavota, and M. Lanza, “Pyref: Refactoring detection in python projects,” in *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2021, pp. 136–141.
- [21] S. Raschka, J. Patterson, and C. Nolet, “Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence,” 2020.
- [22] A. Hamidi, G. Antoniol, F. Khomh, M. D. Penta, and M. Hamidi, “Towards understanding developers’ machine-learning challenges: A multi-language study on stack overflow,” in *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2021, pp. 58–69.
- [23] M. J. Islam, H. A. Nguyen, R. Pan, and H. Rajan, “What do developers ask about ml libraries? a large-scale study using stack overflow,” 2019.
- [24] Y. G, “The 7 steps of machine learning,” Sep 2017. [Online]. Available: <https://towardsdatascience.com/the-7-steps-of-machine-learning-2877d7e5548e>
- [25] S. S. Khairunnesa, S. Ahmed, S. M. Imtiaz, H. Rajan, and G. T. Leavens, “What kinds of contracts do ml apis need?” 2023.
- [26] H. Zhao, Y. Li, F. Liu, X. Xie, and L. Chen, “State and tendency: an empirical study of deep learning question&answer topics on stack overflow,” *Science China Information Sciences*, vol. 64, pp. 1–23, 2021.
- [27] M. Dilhara, A. Ketkar, and D. Dig, “Understanding software-2.0: A study of machine learning library usage and evolution,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 4, pp. 1–42, 2021.
- [28] F. Majidi, M. Openja, F. Khomh, and H. Li, “An empirical study on the usage of automated machine learning tools,” *arXiv preprint arXiv:2208.13116*, 2022.
- [29] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, “Taxonomy of real faults in deep learning systems,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1110–1121.
- [30] R. Zhang, W. Xiao, H. Zhang, Y. Liu, H. Lin, and M. Yang, “An empirical study on program failures of deep learning jobs,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1159–1170.

- [31] V. Riccio, G. Jahangirova, A. Stocco, N. Humbatova, M. Weiss, and P. Tonella, “Testing machine learning based systems: A systematic mapping,” *Empirical Softw. Engg.*, vol. 25, no. 6, p. 5193–5254, nov 2020. [Online]. Available: <https://doi.org/10.1007/s10664-020-09881-0>
- [32] M. M. Morovati, A. Nikanjam, F. Khomh, and Z. M. Jiang, “Bugs in machine learning-based systems: a faultload benchmark,” *Empirical Software Engineering*, vol. 28, no. 3, p. 62, 2023.
- [33] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, “Machine learning testing: Survey, landscapes and horizons,” *IEEE Transactions on Software Engineering*, vol. 48, no. 1, pp. 1–36, 2022.
- [34] M. Wardat, W. Le, and H. Rajan, “Deeplocalize: Fault localization for deep neural networks,” 2021.
- [35] M. Kim, Y. Kim, and E. Lee, “Denchmark: A bug benchmark of deep learning-related software,” in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 540–544.
- [36] M. M. Morovati, A. Nikanjam, F. Tambon, F. Khomh, Z. Ming, and Jiang, “Bug characterization in machine learning-based systems,” 2023.
- [37] R. Guidotti, A. Monreale, S. Ruggieri, F. Turini, F. Giannotti, and D. Pedreschi, “A survey of methods for explaining black box models,” *ACM Comput. Surv.*, vol. 51, no. 5, aug 2018. [Online]. Available: <https://doi.org/10.1145/3236009>
- [38] F. Tambon, A. Nikanjam, L. An, F. Khomh, and G. Antoniol, “Silent bugs in deep learning frameworks: An empirical study of keras and tensorflow,” 2023.
- [39] G. B. de Pádua and W. Shang, “Studying the relationship between exception handling practices and post-release defects,” in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, 2018, pp. 564–575.
- [40] G. B. De Pádua and W. Shang, “Studying the prevalence of exception handling anti-patterns,” in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, 2017, pp. 328–331.
- [41] T. T. Nguyen, P. M. Vu, and T. T. Nguyen, “An empirical study of exception handling bugs and fixes,” in *Proceedings of the 2019 ACM Southeast Conference*, ser. ACM SE

- '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 257–260. [Online]. Available: <https://doi.org/10.1145/3299815.3314472>
- [42] M. Kechagia, M. Fragkoulis, P. Louridas, and D. Spinellis, “The exception handling riddle: An empirical study on the android api,” *Journal of Systems and Software*, vol. 142, pp. 248–270, 2018. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0164121218300724>
- [43] S. Nakshatri, M. Hegde, and S. Thandra, “Analysis of exception handling patterns in java projects: An empirical study,” in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, 2016, pp. 500–503.
- [44] G. B. de Pádua and W. Shang, “Revisiting exception handling practices with exception flow analysis,” 2017.
- [45] A. Schroter, A. Schröter, N. Bettenburg, and R. Premraj, “Do stack traces help developers fix bugs?” in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, 2010, pp. 118–121.
- [46] E. Thiselton and C. Treude, “Enhancing python compiler error messages via stack,” *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 1–12, 2019. [Online]. Available: <https://api.semanticscholar.org/CorpusID:195700198>
- [47] M. Medeiros, U. Kulesza, R. Bonifacio, E. Adachi, and R. Coelho, “Improving bug localization by mining crash reports: An industrial study,” in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 766–775.
- [48] K. K. Sabor, M. Hamdaqa, and A. Hamou-Lhadj, “Automatic prediction of the severity of bugs using stack traces and categorical features,” *Information and Software Technology*, vol. 123, p. 106205, 2020.
- [49] R. Rubel, C. Di Sipio, P. T. Nguyen, J. Di Rocco, and D. Di Ruscio, “Postfinder: Mining stack overflow posts to support software developers,” *Information and Software Technology*, vol. 127, p. 106367, 2020.
- [50] L. Sui, J. Dietrich, and A. Tahir, “On the use of mined stack traces to improve the soundness of statically constructed call graphs,” in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, 2017, pp. 672–676.

- [51] S. Baltes, L. Dumani, C. Treude, and S. Diehl, “Sotorrent,” *Proceedings of the 15th International Conference on Mining Software Repositories*, May 2018.
- [52] H. Borges, A. Hora, and M. T. Valente, “Predicting the popularity of github repositories,” in *Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2016, pp. 1–10.
- [53] —, “Understanding the factors that impact the popularity of github repositories,” in *2016 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2016, pp. 334–344.
- [54] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, “A comprehensive study on deep learning bug characteristics,” in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 510–520.
- [55] G. Nguyen, S. Dlugolinsky, M. Bobák, V. Tran, A. Lopez Garcia, I. Heredia, P. Malík, and L. Hluchý, “Machine learning and deep learning frameworks and libraries for large-scale data mining: a survey,” *Artificial Intelligence Review*, vol. 52, no. 1, pp. 77–124, 2019.
- [56] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, “Taxonomy of real faults in deep learning systems,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1110–1121.
- [57] S. Gupta. (2021) What is the best language for machine learning? [Online]. Available: <https://www.springboard.com/blog/data-science/best-language-for-machine-learning/>
- [58] “Replication: What causes exceptions in machine learning applications? Mining machine learning-related stack traces on stack overflow,” https://github.com/mooselab/ML_stackTrace, 2022.
- [59] J. Zhang, Y. Wang, and D. Yang, “Ccspar: Mining closed contiguous sequential patterns,” *Knowledge-Based Systems*, vol. 89, pp. 1–13, 2015.
- [60] H. Alharthi, D. Outioua, and O. Baysal, “Predicting questions’ scores on stack overflow,” in *2016 IEEE/ACM 3rd International Workshop on CrowdSourcing in Software Engineering (CSI-SE)*, 2016, pp. 1–7.
- [61] M. K. Debbarma, S. Debbarma, N. Debbarma, K. Chakma, and A. Jamatia, “A review and analysis of software complexity metrics in structural testing,” *International Journal of Computer and Communication Engineering*, vol. 2, no. 2, pp. 129–133, 2013.

- [62] S. Xu, A. Bennett, D. Hoogeveen, J. H. Lau, and T. Baldwin, “Preferred answer selection in Stack Overflow: Better text representations ... and metadata, metadata, metadata,” in *Proceedings of the 2018 EMNLP Workshop W-NUT: The 4th Workshop on Noisy User-generated Text*. Brussels, Belgium: Association for Computational Linguistics, Nov. 2018, pp. 137–147. [Online]. Available: <https://aclanthology.org/W18-6119>
- [63] V. Bhat, A. Gokhale, R. Jadhav, J. Pudipeddi, and L. Akoglu, “Min (e) d your tags: Analysis of question response time in stackoverflow,” in *2014 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2014)*. IEEE, 2014, pp. 328–335.
- [64] H. B. Mann and D. R. Whitney, “On a test of whether one of two random variables is stochastically larger than the other,” *The annals of mathematical statistics*, pp. 50–60, 1947.
- [65] Y. Lyu, H. Li, M. Sayagh, Z. M. Jiang, and A. E. Hassan, “An empirical study of the impact of data splitting decisions on the performance of aiops solutions,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 4, pp. 1–38, 2021.
- [66] W. W. LaMorte. (2017) Mann whitney u test (wilcoxon rank sum test). [Online]. Available: https://sphweb.bumc.bu.edu/otlt/mph-modules/bs/bs704_nonparametric/BS704_Nonparametric4.html
- [67] S. H. Khandkar, “Open coding,” *University of Calgary*, vol. 23, p. 2009, 2009.
- [68] P. Burnard, “A method of analysing interview transcripts in qualitative research,” *Nurse education today*, vol. 11, no. 6, pp. 461–466, 1991.
- [69] H. Lune and B. L. Berg, *Qualitative research methods for the social sciences*. Pearson, 2017.
- [70] K.-J. Stol, P. Ralph, and B. Fitzgerald, “Grounded theory in software engineering research: a critical review and guidelines,” in *Proceedings of the 38th International conference on software engineering*, 2016, pp. 120–131.
- [71] K. Krippendorff, “Computing krippendorff’s alpha-reliability,” 2011.
- [72] A. F. Hayes and K. Krippendorff, “Answering the call for a standard reliability measure for coding data,” *Communication methods and measures*, vol. 1, no. 1, pp. 77–89, 2007.

- [73] D. Castelvechi, “Can we open the black box of ai?” *Nature News*, vol. 538, no. 7623, p. 20, 2016.
- [74] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann, “Software engineering for machine learning: A case study,” in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 291–300.
- [75] H. B. Braiek, F. Khomh, and B. Adams, “The open-closed principle of modern machine learning frameworks,” in *Proceedings of the 15th International Conference on Mining Software Repositories*, 2018, pp. 353–363.
- [76] J. Liu, S. Baltes, C. Treude, D. Lo, Y. Zhang, and X. Xia, “Characterizing search activities on stack overflow,” in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 919–931.
- [77] W. Gao, J. Wu, and G. Xu, “Detecting duplicate questions in stack overflow via source code modeling,” *International Journal of Software Engineering and Knowledge Engineering*, vol. 32, no. 02, pp. 227–255, 2022.
- [78] C. Huang, L. Yao, X. Wang, B. Benatallah, and Q. Z. Sheng, “Expert as a service: Software expert recommendation via knowledge domain embeddings in stack overflow,” in *2017 IEEE International Conference on Web Services (ICWS)*, 2017, pp. 317–324.
- [79] B. Kou, Y. Di, M. Chen, and T. Zhang, “Sosum: A dataset of stack overflow post summaries,” in *Proceedings of the 19th International Conference on Mining Software Repositories*, ser. MSR ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 247–251. [Online]. Available: <https://doi.org/10.1145/3524842.3528487>
- [80] “Symptoms of ml application failure,” https://github.com/mooselab/Amin_PR2, 2023.
- [81] P. Ongsulee, “Artificial intelligence, machine learning and deep learning,” in *2017 15th International Conference on ICT and Knowledge Engineering (ICT&KE)*, 2017, pp. 1–6.
- [82] M. M. Morovati, A. Nikanjam, F. Khomh, Z. Ming, and Jiang, “Bugs in machine learning-based systems: A faultload benchmark,” 2023.
- [83] IEEE, “Iso/iec/ieee international standard - systems and software engineering – vocabulary,” *ISO/IEC/IEEE 24765:2010(E)*, pp. 1–418, 2010.

- [84] Developer sentiment around ai/ml. [Online]. Available: <https://stackoverflow.co/labs/developer-sentiment-ai-ml/>
- [85] P. Widjojo and C. Treude, “Addressing compiler errors: Stack overflow or large language models?” 2023.
- [86] A. Antelmi, G. Cordasco, D. De Vinco, and C. Spagnuolo, “The age of snippet programming: Toward understanding developer communities in stack overflow and reddit,” in *Companion Proceedings of the ACM Web Conference 2023*, ser. WWW ’23 Companion. New York, NY, USA: Association for Computing Machinery, 2023, p. 1218–1224. [Online]. Available: <https://doi.org/10.1145/3543873.3587673>
- [87] S. Wang, D. Lo, and L. Jiang, “An empirical study on developer interactions in stackoverflow,” in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, ser. SAC ’13. New York, NY, USA: Association for Computing Machinery, 2013, p. 1019–1024. [Online]. Available: <https://doi.org/10.1145/2480362.2480557>
- [88] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, “Drain: An online log parsing approach with fixed depth tree,” in *2017 IEEE international conference on web services (ICWS)*. IEEE, 2017, pp. 33–40.
- [89] Stack overflow developer survey 2023. [Online]. Available: <https://survey.stackoverflow.co/2023/>
- [90] S. Baltes and P. Ralph, “Sampling in software engineering research: A critical review and guidelines,” *Empirical Software Engineering*, vol. 27, no. 4, p. 94, 2022.
- [91] S. Baltes and S. Diehl, “Usage and attribution of stack overflow code snippets in github projects,” *Empirical Software Engineering*, vol. 24, no. 3, pp. 1259–1295, 2019.
- [92] M. Alshangiti, H. Sapkota, P. K. Murukannaiah, X. Liu, and Q. Yu, “Why is developing machine learning applications challenging? a study on stack overflow posts,” in *2019 acm/ieee international symposium on empirical software engineering and measurement (esem)*. IEEE, 2019, pp. 1–11.
- [93] S. S. Manes and O. Baysal, “How often and what stackoverflow posts do developers reference in their github projects?” in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 235–239.