Demystifying "Bad" Error Messages in Data Science Libraries

Yida Tao
College of Computer Science and Software Engineering
Shenzhen University, China
yidatao@szu.edu.cn

Zhihui Chen
College of Computer Science and Software Engineering
Shenzhen University, China
chenzhihui2019@email.szu.edu.cn

Yepang Liu*
Dept. of Computer Science and Engr.
Southern University of Science and Technology, China
liuyp1@sustech.edu.cn

Jifeng Xuan
School of Computer Science
Wuhan University, China
jxuan@whu.edu.cn

Zhiwu Xu†
College of Computer Science and Software Engineering
Shenzhen University, China
xuzhiwu@szu.edu.cn

Shengchao Qin
College of Computer Sci. & Software Engr., Shenzhen University, China
School of Computing, Engr. & Digital Technologies, Teesside University, UK
shengchao.qin@gmail.com

ABSTRACT

Error messages are critical starting points for debugging. Unfortunately, they seem to be notoriously cryptic, confusing, and uninformative. Yet, it still remains a mystery why error messages receive such bad reputations, especially given that they are merely very short pieces of natural language text. In this paper, we empirically demystify the causes and fixes of "bad" error messages, by qualitatively studying 201 Stack Overflow threads and 335 GitHub issues. We specifically focus on error messages encountered in data science development, which is an increasingly important but not well studied domain.

We found that the causes of "bad" error messages are far more complicated than poor phrasing or flawed articulation of error message content. Many error messages are inherently and inevitably misleading or uninformative, since libraries do not know user intentions and cannot "see" external errors. Fixes to error-message-related issues mostly involve source code changes, while exclusive message content updates only take up a small portion. In addition, whether an error message is informative or helpful is not always clear-cut; even error messages that clearly pinpoint faults and resolutions can still cause confusion for certain users. These findings thus call for a more in-depth investigation on how error messages should be evaluated and improved in the future.

CCS CONCEPTS

• **Software and its engineering → Error handling and recovery**.

*Yepang Liu is also affiliated with Guangdong Provincial Key Laboratory of Brain-inspired Intelligent Computation.
†Zhiwu Xu is the corresponding author.

KEYWORDS

Error message, debugging aid, data science, empirical study

ACM Reference Format:
Yida Tao, Zhihui Chen, Yepang Liu, Jifeng Xuan, Zhiwu Xu, and Shengchao Qin. 2021. Demystifying "Bad" Error Messages in Data Science Libraries. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–27, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3468264.3468560

1 INTRODUCTION

When a program fails, the error message is often the first debugging clue. Research has found that developers allocate a significant portion (up to 25%) of debugging efforts on reading error messages, which is equally demanding as reading source code [4]. Unfortunately, error messages seem to have a bad reputation, as they are often perceived as "cryptic", "unclear", "confusing", "uninformative", "misleading", "unhelpful", "poorly written", and "hard to digest" [3][24][34][35][36][39]. Being the typical diagnostic starting point, poor error messages could substantially prohibit efficient and effective debugging [3][39].

While previous work mostly addressed the quality of error messages generated by compilers [3][4][34], system software [36][39], and static analysis tools [2][20], limited knowledge is available about this issue in the domain of data science. Yet, as data science is experiencing a phenomenal growth in recent years, developing and debugging data science programs have become an indispensable part of modern software engineering [22]. Furthermore, while debugging is generally difficult, debugging data science programs could be doubly challenging when complex data processing, multi-step pipelines and black-box models are heavily involved [17][40]. In such circumstances, poor error messages could only make things worse.

In this work, we present the first empirical study on the quality of error messages encountered when debugging data science programs. Our subjects include six Python libraries (e.g., NumPy, Pandas, and TensorFlow) that are widely used by data science practitioners. We specifically focus on "bad" error messages, since an in-depth understanding of why such error messages are unfavorable and how they are fixed could directly guide the future design of better error messages. Accordingly, we aim to address the following two research questions:

- **RQ1**: Why do certain error messages fail to be helpful debugging aids?
- **RQ2**: How do developers fix "bad" error messages in practice?

For RQ1, we analyzed 201 Stack Overflow (SO for short) threads that contain error messages of the subject libraries in the question descriptions. Intuitively, posting a question on SO is often driven by an unsuccessful debugging attempt. Hence, the inclusion of an error message in the SO question typically indicates that the error message is not helpful enough for debugging, which is exactly the subject we aim to study in RQ1. By comparing these error messages with the actual fixes derived from the accepted SO answers, we characterized the error messages in terms of *whether they pinpoint the faults* and *how they deliver the resolutions*, which essentially measure their capabilities of assisting the *fault localization* and *repair* tasks. We then used this characterization to mechanically distinguish "misleading", "uninformative", and "should be clear" error messages, and investigate the respective underlying causes to answer RQ1.

For RQ2, we analyzed 335 GitHub issues of the subject libraries that are related to error-message quality. By inspecting the thread discussions and associated commits (if any), we derived common strategies library developers adopt to handle such issues. Finally, the findings of RQ1 and RQ2 are aligned to form a comprehensive taxonomy of the causes and fixes to "bad" error messages.

Our study uncovered new knowledge about "bad" error messages in data science libraries, including:

- Around 20% error messages are "misleading" for providing resolutions different from the actual fixes; 61.7% error messages are "uninformative" for being cognitively distant from the actual fixes, even though they indeed point to the right directions for debugging.
- Most "misleading" error messages are in fact logically justifiable from libraries' point of view. Such "misleadingness" can be hard to resolve since libraries are inherently blind to user intentions and external errors.
- Most "uninformative" error messages explicitly describe requirements or violations, which are, however, still cognitively distant from the fix implementations. Yet, due to unclear user intentions or invisible external errors, libraries may not always be able to offer verbatim resolutions in error messages.
- Most error-message issues are eventually fixed by updating source code logics, instead of by updating the phrasing of error messages. The most common fixing strategy is *fail fast*, which adds new checks earlier in the program to prevent late failures with obscure or low-level error messages.
- Library developers and end users could have different interpretations of error messages. An error message that clearly pinpoints the fault and resolution could still be considered as unhelpful by users, especially those who are unfamiliar with the problem domain, the API specification, or the programming language.

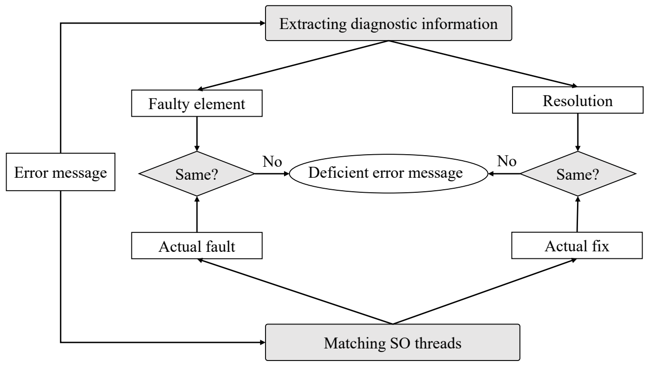


Figure 1: The workflow of identifying "bad" error messages.

In general, "bad" error messages are not necessarily equivalent to bad phrasing or presentation of error messages. Our study revealed deeper causes to error message deficiency, which is fundamentally attributed to *how much information libraries can access* and *how responsible libraries should be in terms of being diagnostic and prescriptive*. We hope that our findings could raise awareness of these underlying reasons that lead to "bad" error messages, and therefore motivate practitioners to design and consume error messages more effectively. The code and data of this study are publicly available [33].

2 METHODOLOGY

In this section, we introduce our methodology in detail.

2.1 Data Collection

The subjects of this study include six popular libraries from the Python ecosystem for data science. Specifically, NumPy [8] and Pandas [26] provide flexible functionalities to manipulate structured data; SciPy [9] and Scikit-learn [28] feature various numerical routines and algorithms for statistics and machine learning; TensorFlow is a general-purpose deep learning framework [1]; Gensim is a library for unsupervised topic modeling and natural language processing [30]. Together, these libraries can be used to solve different types of important tasks in data science.

We first identified all the error messages that can be potentially generated by subject libraries. To this purpose, we recursively invoked Python's `dir()` function to explore all methods in the subject modules. During the process, we collected the API signatures of these methods (e.g., `pandas.crosstab`) and the respective source code using Python's introspection capability [14]. We then built abstract syntax trees from the source code and traversed the AST to collect all nodes with the type `ast.Raise`, which represents the statement where an error is explicitly raised (thrown). The corresponding error messages can in turn be retrieved from the `Raise` nodes. Using this approach, we identified 4,744 error messages from the subject libraries, as shown in Table 1.

Table 1: Dataset statistics.

Library	Version	# Error messages	# Error messages appeared in SO	# Error messages analyzed in SO context	# Error-message issues with confirmed fixes
NumPy	1.18.5	197	37 (19%)	18	55
Pandas	1.0.3	503	183 (36%)	62	105
SciPy	1.4.1	1,454	280 (19%)	32	28
Scikit-learn	0.22.1	523	125 (24%)	44	52
TensorFlow	2.1.0	1,881	225 (12%)	42	37
Gensim	3.8.0	186	40 (22%)	3	3
Total		4,744	890 (19%)	201	280

2.2 Characterizing Deficient Error Messages

To address RQ1, error messages should be evaluated in practical debugging contexts with known resolutions. We used the SO platform for this purpose. As mentioned in Section 1, if a user posted an SO question that is associated with an error message, we might reasonably assume that s/he is having difficulty in debugging even in the presence of the error message. In other words, the error message is potentially insufficient to aid debugging, which is exactly the subject we aim to study. In addition, since SO strongly advocates the reproducibility of the proposed problems and resolutions [10], we could study target error messages in self-contained, reproducible, and practical debugging contexts. The reward system of SO further allowed us to identify known resolutions from the accepted answers with sufficient confidence.

Figure 1 shows the workflow of this process. Basically, two types of diagnostic information, faulty element and resolution, are extracted from error message content and cross-checked against the actual fixes in accepted SO answers. Error messages that do not pinpoint the same faulty elements or resolutions were considered as deficient, and were manually analyzed to identify the underlying causes. Below we describe each step in detail.

2.2.1 Matching SO Threads. Using the official Stack Exchange REST API [11], we collected 299,181 SO threads tagged with the subject libraries. We then searched for SO questions that contain the extracted error messages and also have the respective library tags. A notable step here is the preprocessing of string formatting, which is often used in error messages to capture dynamic information (e.g., input values observed at runtime). Major approaches to Python string formatting include the C-style formatting (e.g., `"Shape mismatch: %s" % var`), string.format (e.g., `"Shape mismatch: {}".format(var)`), and f-strings (e.g., `f"Shape mismatch: {var}"`) [13]. To construct proper search queries for formatted error messages, we replaced the placeholders in these strings with wildcards so that the fixed part (e.g., "`Shape mismatch:`") will be matched while the variable part (e.g., `var`), which may vary across different debugging contexts, will be ignored.

Using this approach, we identified 890 error messages that appeared in SO questions, which account for nearly one fifth of the total error messages extracted from the subject libraries (Table 1). We then retained error messages whose SO questions have positive upvotes and accepted answers for further analysis. Note that if an error message appeared in multiple eligible SO questions, we selected the question with more upvotes. SO threads were skipped if we had difficulty understanding the debugging contexts. Finally, the SO threads of 201 distinct error messages (Table 1) were analyzed to address RQ1.

2.2.2 Evaluating Error Message Content. We evaluate the quality of the 201 error messages from the following two aspects.

Whether the error message pinpoints the exact faulty element: We use *faulty elements* to refer to code elements, such as parameters, variables, types, and operations, that cause the error and should be fixed. To assist the identification of such elements, which are essentially names, we used the Stanford CoreNLP toolkit [23] to extract words from error messages with the NN (noun), NNP (proper noun) and NNS (noun, plural) Part-of-Speech tags. We then cross-checked the accepted SO answer to determine whether the faulty elements identified from the error message were the same as those addressed in the actual fix, which is essentially a binary labeling process.

Consider the example #1 in Table 2. Since the fault `ignore_index` addressed in the accepted fix also appears as a noun in the error message, we assigned a positive label for this message for pinpointing the exact same faulty element. On the other hand, the error message in the example #5 received a negative label for pinpointing a faulty element (i.e., *the 'sep' keyword*) that is different from the one being actually fixed (i.e., *the DataFrame df*).

How the error message delivers the resolution: We determined the resolution provided in an error message using the following heuristics. First, if the error message contains words such as "must", "should", "need", "have to", "required", and "expected", we consider the content entailed by these words as the suggested resolution (e.g., #3 in Table 2). Otherwise, we looked for negative words such as "unexpected", "cannot", "unable", and "invalid", and used the opposite meaning of the content entailed by these words as the suggested resolution. For example, the error message *"index is not a valid DatetimeIndex or PeriodIndex"* insinuates that index should be a valid `DatetimeIndex` or `PeriodIndex` (#2 in Table 2).

We then labeled whether the resolution identified from the error message is the same as the one implemented in the actual fix. In Table 2, #1 and #2 are positive examples of this task. For those with negative labels, however, we observed a special group of error messages, which are distant from the actual fix but still point to the right direction (e.g., #3 and #4 in Table 2). Given that error messages are used as debugging aids, the ones that point to the

Table 2: Examples of error message characterization.

#	Error message	Actual Fix	Faulty element	Delivered resolution	Perception
1	*Can only append a Series if ignore_index=True or if the Series has a name.*	Add `ignore_index=True` when calling `append()`. (so-33094056)	Same	Same	Clear
2	*index is not a valid DatetimeIndex or PeriodIndex.*	Convert the index to the `DatetimeIndex` type. (so-26089670)	Same	Same	Clear
3	*The score function should be a callable, 26.48 was passed.*	Import `chi2` right before assigning it to the `score` function to avoid accidental overwritting. (so-60253980)	Same	Indirect	Uninformative
4	*Failed to convert object of type <class 'dict'> to Tensor .... Consider casting elements to a supported type.*	Convert the `y` labels to the `pd.Series` type. (so-54668364)	Different	Indirect	Uninformative
5	*Did you mean to supply a 'sep' keyword?*	Use `df[['a','b']].values` instead of `df[['a','b']]`. (so-53054127)	Different	Different	Misleading
6	*Table sqlite:///C:tmp.db not found.*	Pass a `sqlalchemy` engine object instead of a string to `pd.read_sql_table`. (so-33083415)	Different	Different	Misleading

right direction should be differentiated from the ones that point to the wrong direction (e.g., #5 and #6 in Table 2). For this concern, we assigned a special label, *indirect*, to such error messages.

The manual labeling was performed by the first two authors independently. The Cohen's kappa for their labelings in the two tasks are 0.78 and 0.67, respectively, indicating a substantial inter-rater agreement [25]. Disagreements were reconciled with the third author joining the discussions.[1]

2.2.3 Understanding Deficiency Causes. With these labelings, we were now able to concretize the "badness" of error messages. First, error messages that provide *different* resolutions could be perceived as "misleading". Then, error messages that provide *indirect* resolutions or pinpoint *different* faulty elements could be considered as "uninformative". Finally, error messages that describe both the same faulty elements and resolutions as the actual fixes should be considered as "clear". Table 2 shows examples of each category.

We then manually categorized the underlying causes to misleading and uninformative error messages. As no prior knowledge is available for this type of categorization, we adopt the *inductive coding* method in the process [5]. Initially, the first two authors independently inspected the SO threads and related artifacts (e.g., source code, documentation, etc.) and created tentative codes that best summarize the deficiency causes. Then, they got together to compare the independent codes, merge similar concepts, and reconcile disagreements. This process continued until the categories of all deficiency causes were agreed by both annotators. We report the findings in Section 3.

2.3 Characterizing Error Message Fixes

To address RQ2, we studied subject libraries' GitHub issues to analyze developers' strategies of handling error message issues. Specifically, we searched for *closed* issues whose descriptions contain either the "error message" string or the text of the 201 error messages studied in RQ1. Intuitively, the search criteria help to identify issues that address users' complaints on error messages, and we are likely to identify a definite reaction from developers for an issue if it is already marked as *closed*.

Using the official GitHub REST API [16], we collected 1,026 GitHub issues that satisfied the search criteria. To make the inspections manually feasible and statistically meaningful, we randomly sampled half of the issues. During the process, we discarded issues that do not really address error message quality (e.g., gensim-gh-235[2]) or address only trivial typos and spacing problems in error messages. We also discarded issues whose duplicate or related issues have already been inspected.

Finally, 335 issues were considered valid. Among which, 280 have associated commits that were merged to the main branch, indicating that the issues were confirmed and fixed by developers. For these issues (Table 1), we inspected both their thread discussions and merged commits to categorize library developers' strategies of fixing deficient error messages. We also studied the remaining 55 valid issues to understand why they were closed without any change being made to the source code. An inductive coding approach similar to the one described in Section 2.2.3 was also adopted in this process. The findings are reported in Section 4.

3 RQ1: WHY DO CERTAIN ERROR MESSAGES FAIL TO BE HELPFUL DEBUGGING AIDS?

Based on the characterization described in Section 2.2, we identified 19.9% *misleading* error messages and 61.7% *uninformative* error messages in our dataset (Table 3). We also identified 18.4% error messages that are supposedly *clear* but still being asked on SO. In this section, we report the underlying causes to the deficiency of these error messages.

[1]The number of disagreements for the two labeling tasks are 18 and 39, respectively.

[2]Throughout the paper, we use so-questionID to uniquely identify an SO post (https://stackoverflow.com/questions/questionID), and library-gh-issueID to uniquely identify a GitHub issue (https://github.com/library/issues/issueID).

Table 3: Error message evaluation results.

Faulty Element	Resolution			
	Same	Indirect	Different	Total
Same	37	96	16	149 (74.2%)
Different	-	28	24	52 (25.8%)
Total	37 (18.4%)	124 (61.7%)	40 (19.9%)	201 (100%)

3.1 Misleading Error Messages

As explained in Section 2.2.3, an error message is likely to be considered as “misleading” if it suggested a resolution that is different from the actual fix. Table 3 shows that there are 40 (19.9%) error messages of such kind. Surprisingly, among these 40 error messages, we identified only one error message from SciPy that is misleading indeed because it is poorly phrased. We made a commit to rephrase this error message and submitted a corresponding pull request, which was confirmed by SciPy developers and merged to the library code. [3]

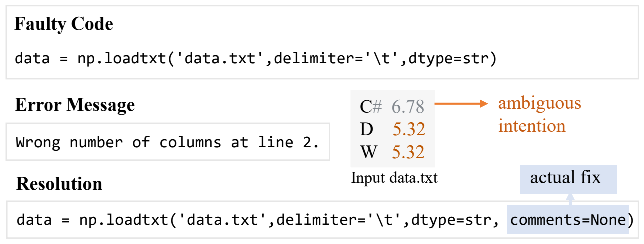
Nonetheless, for the remaining 39 error messages, the underlying causes to their “misleadingness” are more sophisticated than how the messages themselves are phrased. Below we present our observations.

3.1.1 Library is blind to user intentions or errors (20). Users can use a library in many different ways. Yet, from the perspective of library developers, how users intend to use the library is often ambiguous. Consequently, libraries might produce error messages that are logically justifiable but still misleading to users.

Consider numpy-so-28254992, which reports the error “*Wrong number of columns at line 2*” when `numpy.loadtxt` is invoked to load a text. This error occurred since the “#” symbol in the first line of the input data is by default interpreted as a comment, yet the user actually intended to use “#” as a normal symbol in the word “C#” (Figure 2). Therefore, the accepted resolution is to pass `comments=None` to `numpy.loadtxt` to disable the default behavior, instead of fixing the number of columns in the input text as suggested by the error message.

While the error message sounds misleading in this particular case, it would also make perfect sense if users indeed intend to use “#” as comments in the input. From the perspective of library developers, it would be difficult to distinguish between a deliberate and a careless API usage, especially when the default behavior is explicitly described in the official documentation.[4] In fact, this error message was discussed in numpy-gh-1810 and numpy-gh-2591, in which developers decided that for cases where an input line (e.g., line 2 in the example) has different columns than its previous line, the error message will only spit out the current line number since “*it is not always clear where the problem is*”. [5]

In addition to user intentions, a library is also blind to external errors that are essentially inaccessible from within the library’s namespace. For such cases, it would be impractical to expect the library to raise an error message that accurately describes the resolutions. Figure 3 shows an example from pandas-so-40533647. Here,



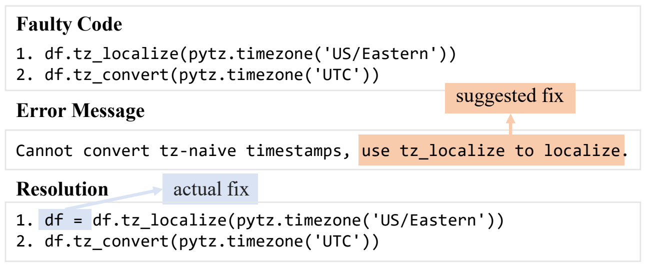
```
data = np.loadtxt('data.txt',delimiter='\t',dtype=str)
```

```
Wrong number of columns at line 2.
```

```
C# 6.78
D  5.32
W  5.32
```

```
data = np.loadtxt('data.txt',delimiter='\t',dtype=str, comments=None)
```

Figure 2: Example of ambiguous user intention.



```
1. df.tz_localize(pytz.timezone('US/Eastern'))
2. df.tz_convert(pytz.timezone('UTC'))
```

```
Cannot convert tz-naive timestamps, use tz_localize to localize.
```

```
1. df = df.tz_localize(pytz.timezone('US/Eastern'))
2. df.tz_convert(pytz.timezone('UTC'))
```

Figure 3: Example of external user errors.

`tz_convert` failed at line 2 and threw an error message that suggests “*use tz_localize to localize*”, which may sound confusing since the user already did so at line 1. The real fault, however, was that the user did not assign the computation result at line 1 back to `df`, hence the `tz_localize` operation did not really take effect at line 2. Yet, the library could not possibly “see” this external error. In this sense, the error message was logically justifiable for explicitly demanding the missing `tz_localize` operation.

In general, half of the inspected “misleading” error messages seem to be logically reasonable from the library’s perspective. The primary reason for their “misleadingness” is that libraries are inherently unaware of user intentions or user errors, which restrict their abilities of offering accurate resolutions in error messages.

3.1.2 Library is buggy or outdated (16). When errors were caused by library bugs or incompatibilities, accepted resolutions were typically updating library versions or applying workarounds. Therefore, error messages that did not suggest the same might be considered as misleading. For example, pandas-so-53054127 reports an error “*Did you mean to supply a ‘sep’ keyword?*”. As explained in the accepted answer, the error was due to a bug in Pandas 0.21.1, which was fixed in 0.23.4. Users could either update the library version or apply a workaround to resolve the error (#5 in Table 2), instead of *supplying a ‘sep’ keyword* as suggested by the error message.

Take pandas-so-33083415 as another example. An error “*Table sqlite:///C:tmp.db not found*” occurred because Pandas 0.16 was installed to invoke `read_sql_table` with a string URL, which was only supported since version 0.17. The accepted resolution was either conforming the code to the 0.16 specification (#6 in Table 2) or simply updating the library version, which had nothing to do with the *table existence* insinuated by the error message.

3.1.3 Open design issues (3). Finally, we identified three cases where the error messages demand consistent *input shapes*, which

[3]https://github.com/scipy/scipy/pull/13119
[4]https://numpy.org/doc/stable/reference/generated/numpy.loadtxt.html
[5]https://github.com/numpy/numpy/issues/1810#issuecomment-9619267

was different from the *input sparsity* issue tackled in the actual fixes. For example, numpy-so-41349326 reports an error "*all input arrays must have the same shape*" when invoking `numpy.stack`. However, what really caused this error was that a *sparse* matrix that was not recognized by the API was used as input.

In fact, the shape of sparse matrices is "*a known and open issue*" in the Python data science community.[6] Even library developers themselves have described this issue as "*a general cause of confusion*" and "*an endless source of bugs*", and "*trying to treat np.matrix, np.ndarray, scipy.sparse, or masked matrices through a common interface is a lost cause*",[7] not to mention end users who may not even be familiar with all these concepts.

Finding 1: Due to ambiguous user intentions, external user errors, and open design issues, libraries can inevitably produce error messages that are misleading to users.

3.2 Uninformative Error Messages

In addition to "misleading" error messages, error messages that do not pinpoint the same faulty elements or provide only indirect resolutions are also worth exploring as they might be considered as "uninformative".

3.2.1 Error messages do not pinpoint the same faulty elements. As shown in Table 3, 52 (25.8%) error messages do not pinpoint the same faulty elements. Among them, 24 also provide resolutions different from the actual fixes, which were already discussed in Section 3.1. For the remaining 28 error messages, we identified two distinctive deficiency patterns.

Faulty element is too general (4). In this type of error messages, the faulty element being referred to is too general to be helpful. For example, pandas-so-34532954 reports the error message "*DataFrame constructor not properly called*". The message was not wrong, given that the first argument passed to the constructor indeed had an incorrect type. However, the constructor of `pandas.DataFrame` has five parameters, each has several valid options, and together many combinations are possible. In such cases, knowing the error message is still insufficient for locating the specific fault.

We observed that two of these error messages have already been fixed by library developers later on, and the fixes indeed changed error messages to be more specific on the faulty elements.[8] For the other two error messages (e.g., "*Domain error in arguments*"), we submitted corresponding issues addressing their uninformativeness, which have been confirmed by library developers.[9]

Faulty element is too low-level (24). Most of the error messages that do not pinpoint the same faulty elements in fact pinpoint the final *manifestations* of the fault. For example, tensorflow-so-42888277 discussed a case where the user forgot to set the `label_dimension` parameter when invoking the `DNNRegressor` API. This caused the program to crash with an overwhelmingly long stack trace ended with the message "*Shapes (118, 1) and (118, 3) are incompatible*". Here, the "incompatible shape" pinpointed by the error message was essentially the final manifestation of the missing parameter.

[6]https://github.com/numpy/numpy/issues/7782#issuecomment-229310312
[7]https://github.com/scipy/scipy/issues/4239#issuecomment-65922166
[8]The fixing commits are 3ec9d6 for numpy and 2dd80c for pandas.
[9]Issues #13172 and #13173 for SciPy.

For end users, this type of error messages might be considered as less straightforward since they pinpoint internal errorneous status resulted from the fault instead of the fault itself. In this sense, their "uninformativeness" is more of a byproduct of where the error is handled, rather than how the error message itself is phrased. In Section 4.1.1, we discuss how library developers handle such cases in practice.

Finding 2: The faulty elements pinpointed by error messages could be too low-level (internal) or too high-level (general) to be informative. The former case is more commonly observed.

3.2.2 Error messages provide indirect resolutions. Table 3 shows that 124 (61.7%) error messages suggest indirect resolutions. Among which, 28 were indirect also for pinpointing different faulty elements, as discussed in Section 3.2.1. For the remaining 96 error messages, we observed the following two deficiency causes.

Requirements are hardly derivable from the error messages (4). This type of error messages tends to describe what is violated or not allowed. However, what is required is hardly derivable from the violations. For example, the error message "*Not supported for type Index*" (pandas-so-45189650) states that `Index` is not supported; yet, it is still unclear which types are supported. For users who expect a direct answer, such error messages might be regarded as uninformative. To improve these error messages, we identified their respective requirements from the official documentation and submitted four pull requests. The one that fixes the error message in the prior example was already merged at this moment.[10] Specifically, our merged fix changed the message to "*This method is only implemented for DatetimeIndex, PeriodIndex and TimedeltaIndex; Got type Index*", which became more actionable by stating explicit requirements.

Requirements in error messages are explicit, but still cognitively distant from the fix implementations (92). Even if error messages explicitly state the requirements, there may still exist nontrivial cognitive gaps between understanding the requirements and implementing the final fixes. For example, in pandas-so-36519546, `pandas.to_numeric(df)` raised the error "*arg must be a list, tuple, 1-d array, or Series*" since `df` was of an unexpected type. The accepted fix was to use `to_numeric` as the lambda function when invoking another API `pandas.apply` (i.e., `df.apply(lambda x: pd.to_numeric(x), axis=0)`), which indeed complied with the error message by passing a `Series` argument to `to_numeric`. Yet, implementing this fix could still be cognitively demanding, especially for users who were unfamiliar with the `pandas.apply` API or the lambda function.

Cognitive gaps are sometimes also the inevitable consequences of unknown user intentions and errors, which are practically invisible from the libraries. For example, the error message shown in Table 2 #3 explicitly required "*the score function to be a callable*". However, nontrivial debugging efforts were still needed for users to figure out why this requirement was violated, which in this case was due to an accidental overwritting of `score` in the client code. Since this error was not even visible from the library's namespace, the error message could only highlight the concept of mistake instead of offering direct resolutions.

[10]https://github.com/pandas-dev/pandas/pull/38176

Finding 3: For error messages that offer indirect resolutions, explicit requirements are either missing or cognitively distant from the fix implementations. However, it is not always possible for error messages to provide verbatim resolutions, also due to unknown user intentions and errors.

3.3 Clear But Still Unhelpful Error Messages

Finally, error messages that pinpoint the same faulty elements and resolutions as the actual fixes should be much clearer than the ones discussed above. Interestingly, as shown in Table 3, such error messages were still observed in SO threads (18.4%), meaning that users knowing the messages were still stuck in debugging regardless. We identified the following reasons for this phenomenon.

3.3.1 Insufficient domain-specific knowledge. First, users with insufficient domain knowledge may have a hard time understanding error messages even when they are relatively clear. For example, in sklearn-so-27623370, a user used a test data of 12 features on a machine learning model that was trained on data with two features, resulting in the error "*Number of features of the model must match the input. Model n_features is 2 and input n_features is 12*". In this case, the error message directly pinpoints the fault (i.e., the number of features) and the resolution (i.e., the features must match). However, the user was still stuck, possibly due to his/her insufficient knowledge on the basic concepts in training and testing machine learning models.

3.3.2 Unfamiliarity with API specification. A clear error message may still be considered unhelpful if users were unfamiliar with the API specification in the first place. For example, in sklearn-so-54980098, a user received the error "*x is neither increasing nor decreasing*" when passing an array of labels to `sklearn.metrics.auc`, whose official documentation explicitly states that the argument must be an x coordinate that is either monotonically increasing or decreasing. In this case, the error message should be clear in terms of both the faulty element and the resolution if users are familiar with the API specification.

3.3.3 Unfamiliarity with the programming language. Finally, users' programming expertise could also affect their perceptions of error messages. For example, the `scipy.optimize.check_grad (func, grad, x0, *args, **kwargs)` API allows an arbitrary number of arguments, indicated by `*args` and `**kwargs`. In scipy-so-40286648, a user incorrectly used `arg` as if it was a normal keyword argument, and therefore received the error message "*Unknown keyword arguments: ['args']*". Even after the user checked the documentation of this API, s/he was still convinced that `args` was a valid argumet and that the error should not occur (the user asked: "*Noteworthy args is listed as an argumet in the help file. Shouldn't this work?*"). In this case, if the user was familiar with Python's syntax on this type of variadic arguments (e.g., `*args`), the error message itself should be self-explanatory.

Finding 4: Error messages that pinpoint the exact faulty elements and resolutions might still be considered as unhelpful, especially when users are unfamiliar with the problem domain, the API specification, or the programming language.

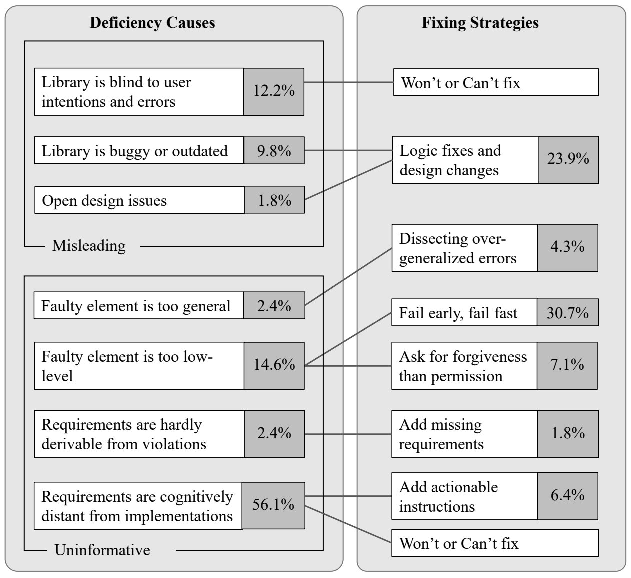

Deficiency Causes
Fixing Strategies
Library is blind to user intentions and errors
12.2%
Library is buggy or outdated
9.8%
Open design issues
1.8%
Misleading
Faulty element is too general
2.4%
Faulty element is too low-level
14.6%
Requirements are hardly derivable from violations
2.4%
Requirements are cognitively distant from implementations
56.1%
Uninformative
Won't or Can't fix
Logic fixes and design changes
23.9%
Dissecting over-generalized errors
4.3%
Fail early, fail fast
30.7%
Ask for forgiveness than permission
7.1%
Add missing requirements
1.8%
Add actionable instructions
6.4%
Won't or Can't fix

Figure 4: Deficiency causes and respective fixing strategies.

4 RQ2: HOW DO DEVELOPERS FIX "BAD" ERROR MESSAGES?

In this section, we report developers' practice on handling deficient error messages. Figure 4 shows an overview of the fixing strategies and their associations with the deficiency causes described in Section 3. [11]

4.1 Fixing Strategies

To study fixing strategies, we analyzed 280 GitHub issues that address error message quality and have confirmed fixes (Section 2.3). Table 4 shows the types of artifacts changed in these fixes. Interestingly, although error messages are being complained in these issues, only 27.9% fixes exclusively update the message content. The majority of fixes, however, involve updates to the source code. In particular, 56.4% fixes only update the source code without even changing the error message content. Below we discuss detailed fixing strategies with respect to source code changes and error message changes.

4.1.1 Source code changes. We identified four major strategies developers adopt when they change source code to handle deficient error messages.

Fail early, fail fast (30.7%): The majority of fixes adopt the "*fail fast*" principle, which states that software should fail *immediately* and *visibly* when a problem occurs [32]. Fixes adopting this strategy typically add explicit checks earlier in the program without even modifying the error message content.

Consider pandas-gh-3481, in which the error "*Cannot call bool() on DataFrame*" was raised on `pandas.concat(df1,df2)`. The user complained that the message was "seriously cryptic", given that

[11]The percentage of deficiency causes in Figure 4 is calculated by dividing the occurrence of each cause by 164, which is the number of error messages that do not provide the same faulty elements and resolutions (Table 3).

Table 4: Types of changes made to fix deficient error messages. Less than one third fixes changed message content exclusively.

Error message changed	Source code changed	Total
✓	✗	78 (27.9%)
✗	✓	158 (56.4%)
✓	✓	44 (15.7%)

```
  def __init__(self, objs, axis=0, join='outer', join_axes=None,
               keys=None, levels=None, names=None,
               ignore_index=False,  verify_integrity=False):
+   if not isinstance(objs, (tuple, list, dict)):
+       raise AssertionError('first argument must be a list of pandas '
+                            'objects, you passed an object of type '
+                            ' "{0}" '.format(type(objs).__name__))
```

Figure 5: Example of the "fail fast" fixing strategy.

the actual fix should be `pandas.concat([df1,df2])`. Library developers also confirmed this issue and commented "*should fail fast*". As shown in Figure 5, the fix to this issue was adding code to check whether the objects being concatenated are in a list as soon as they were passed to the API. With this fix, a more meaningful error message, "*first argument must be a list of pandas objects, you passed an object of type 'DataFrame'*", will be raised immediately for the same API misuse.

Recall the findings from Section 3.2.1, which state that error messages with low-level details were commonly observed and potentially uninformative. The *fail fast* strategy could effectively address this type of deficiency, as it prevents a program from proceeding silently when an error occurs and crashing strangely later on with obscure internal errors. In addition, by adding error handling code earlier and closer to where the error occurs, critical context information such as the faulty element becomes accessible from within the library's namespace and thus could be better embedded in the error messages.

Ask for forgiveness than permission (7.1%): This strategy basically wraps an existing code with a `try` block and raises a more informative error message in the corresponding `except` block. Consider pandas-gh-26554, in which the user complained that `from_coo` gave a "not so useful" error message when given an input of non-coo matrix type. As shown in Figure 6, the fix was to wrap the corresponding statement in a `try. . . except` block with a more user-friendly error message "*Expected coo_matrix. Got csr_matrix instead*".

In contrast to *fail fast*, which is essentially a preventive strategy, this second strategy adopts the principle of "ask for forgiveness than permission" [12]. To be more specific, this strategy still let whatever internal errors to occur, but then catch them at the code of interest with more context-aware error messages that mask low-level and obscure details.

Dissecting overgeneralized errors (4.3%): While the previous two strategies avoid low-level error messages, this third strategy improves overgeneralized error messages by adding necessary helper code. Take sklearn-gh-344 as an example. Initially, a general error

```
-  s = Series(A.data, MultiIndex.from_array((A.row, A.col)))
+  try:
+      s = Series(A.data, MultiIndex.from_arrays((A.row, A.col)))
+  except AttributeError:
+      raise TypeError('Expected coo_matrix. Got {} instead.'
+                 .format(type(A).__name__))
```

Figure 6: Example of the "ask for forgiveness" strategy.

```
  if not solver_type in self._solver_type_dict:
+     if self.penalty.upper() == 'L1' and self.loss.upper() == 'L1':
+         error_string = ("The combination of penalty='l1' "
+             "and loss='l1' is not supported.")
+     elif self.penalty.upper() == 'L2' and self.loss.upper() == 'L1':
+         error_string = ("loss='l2' and penalty='l1' is "
+             "only supported when dual='true'.")
+     else:
+         error_string = ("penalty='l1' is only supported "
+             "when dual='false'.")
      raise ValueError('Not supported set of arguments: '
-                      + solver_type)
+                      + error_string)
```

Figure 7: Example of dissecting overgeneralized errors.

"*Not supported set of arguments*" was raised, which was considered by users as "not clear at all". Figure 7 shows the fix to this issue, which added conditional checks to explore different erroneous combinations of parameters. As a result, the original overgeneralized error messages was now dissected to three different error messages with respect to three subconditions of errors (e.g., *penalty='l1' is only supported when dual='false'*).

Logic fixes and design changes (23.9%): While the aforementioned strategies focus on general error handling practice, a non-trivial proportion of fixes also address project-specific logic flaws or design issues. For example, an unhelpful error message "*data not found*" was complained in scipy-gh-7718. The respective fix was to defer a dictionary initialization to after an input type check, so that an existing error message "*Save is not implemented for sparse matrix of format dok*", which is more helpful, could be reached and thrown first. As another example, tensorflow-gh-6035 discussed a "very obscure" error message "*No OpKernel was registered to support Op 'MaxPoolWithArgmax' with these attrs. Registered devices: [CPU]*", when `tf.nn.max_pool_with_argmax` was executed on CPU but only implemented for GPU. To fix this issue, developers made a major design change by adding a CPU support to the `max_pool_with_argmax` op.

Finding 5: Source code changes in error-message-related fixes typically implement more robust error handlings. Project-specific logic fixes or design improvements could also prevent deficient error messages from being raised.

4.1.2 *Error message content changes.*

We identified nine strategies developers adopt when they change error message content to address related complaints. Table 5 describes these strategies

Table 5: Strategies for fixing error message content.

Focus	Strategy	Example of error message changes (added, ~~deleted~~)
Completeness	Add runtime contexts (12.9%)	“k=%d must be between 1 and rank(A)-1=%d” % (k, n-1) (scipy-ebb8b1)
	Add actionable instructions (6.4%)	“X needs to contain only non-negative integers. Please set categories=’auto’ explicitly to be able to use arbitrary integer values as category identifiers.” (sklearn-94c70f)
	Add missing requirements (1.8%)	“method must be either ’pearson’, ’spearman’, ~~or~~ ’kendall’, or a callable.” (pandas-64f596)
Clearness	Clarification (7.5%)	“min_samples_split must be ~~at least 2~~ an integer greater than 1 or a float in (0, 1.0].” (sklearn-84cc67)
	Rephrase (4.6%)	“If there is no initial_state ~~is provided~~, you must give a dtype ~~must be~~.” (tensorflow-06be84)
Correctness	Fix incorrect info (6.1%)	“The length of the input vector x must be ~~at least~~ greater than padlen.” (scipy-be94e5)
	Fix outdated info (2.1%)	“Use with ~~default_session(sess)~~ sess.as_default() or pass an explicit session to run(session=sess).” (tensorflow-3e7aae)
Succinctness	Remove internal details (2.1%)	“{} ~~are~~ dtype not supported ~~in cython ops~~.” (pandas-6b23fb8d)
	Reduce message length (1.1%)	“~~Valid options are [’accuracy’, ’adjusted_mutual_info_score’…(a very long list is omitted)]~~ Use sorted(sklearn.metrics.SCORERS.keys()) to get valid options.” (sklearn-825cf0)

with examples. In general, these strategies focus on improving the following aspects of deficient error messages.

Completeness (21.1%): The majority of error message content changes add new information to improve completeness. As shown in Table 5, the most common type of information being added is the *runtime context* information. Specifically, relevant variable values or computation results observed when an error occurs are often added to the corresponding error message, so that users could better understand the errorneous program state and pivot debugging.

Actionable instruction is another type of information that is often added to complete error messages. Such information might effectively reduce the cognitive workload for implementing fixes (see Section 3.2.2). We also observed a few cases where crucial requirements that were previously missing were added to complete the error messages.

Clearness (12.1%): Error messages with complete information may still be considered deficient if the presentation is unclear. Developers fix such error messages by clarifying concepts or rephrasing sentences as shown in Table 5.

Correctness (8.2%): In addition, developers update error message content to ensure that they are correct and up-to-date.

Succinctness (3.2%): In a few cases, developers remove unnecessary or verbose information from error messages to keep them concise and readable.

Finding 6: Adding runtime context information is the most common strategy for improving error message content.

4.2 Won’t Fix

As described in Section 2.3, we identified 55 error message related issues that were closed without any merged commits. Below we summarize salient reasons why library developers hesitate to confirm or address these issues.

Conceptual mismatch between library developers and users: Since library developers and end users are not equally familiar with the design or implementation of the library, they might have different interpretations of the same error messages. Consider the error message “*Unalignable boolean Series key provided*” discussed in pandas-gh-14491. A user considered this error message to be “very confusing” since the term “alignable” intuitively refers to matching *lengths*. On the contrary, a library developer considered this message to be “rather clear”, since “alignable” naturally denotes the matching of both *lengths* and *labels* “in the pandas-world”.

Numpy-gh-13666 is another example, in which a library developer considered the error message “*loop of ufunc does not support argument 0 of type int which has no callable sqrt method*” to be reasonably clear for “trying to explain what is going on”. However, a user commented that “*the error message may be helpful for numpy developers but is still cryptic for numpy users who have no idea what ufunc is or which loop it’s referring to.*”

Issues are out of scope: Developers also turn down requests to fix error messages when problems are not really related to the libraries. For example, the error message “*merge() takes at least 2 arguments (3 given)*” was considered unhelpful in pandas-gh-18528. Yet, library developers pointed out that this error was actually a de facto Python compiler behavior for checking incorrect argument passing, which occurred “*before pandas even sees the call*”.

In numpy-gh-7782, a user proposed to improve a misleading error message by adding explanations on matrix sparsity. However, developers turned down this proposal since the concept of “sparse matrix” is only used in SciPy and is not introduced in NumPy.

User intentions are unknown: While users may prefer *prescriptive* error messages that offer direct resolutions, developers sometimes prefer *descriptive* error messages since libraries are blind to user intentions. For example, in pandas-gh-25313, a user proposed to replace an error message “*KeyError: 1.75*” with “*Value was not found in index. To match the closest value, use the parameter method=‘nearest’*”. A library developer commented that “*I’m not really fond of being prescriptive with solution as pandas is blind to the user’s intentions*”. This observation is also consistent with our findings in Section 3.2.2, which point out that error messages sometimes can only describe the mistakes instead of providing verbatim resolutions due to unknown user intentions.

Finding 7: Error messages that are considered deficient by end users might be perceived differently by library developers.

5 IMPLICATIONS

In this section, we summarize the implications of our findings from the perspective of different stakeholders.

For library users, their intended usages sometimes are inherently unknown to the libraries. Hence, error messages generated by libraries *cannot* always offer direct explanations or resolutions as expected. With this awareness, library users should avoid overreliance on error messages in debugging. Instead, they should be more attune to other reasoning processes, such as exploring debugging contexts, re-examining problem definitions, inspecting library documentation, or even revisiting programming language specifications, during debugging.

For library developers, more robust error handlings should be practiced to prevent internal error messages from being raised and users from being overwhelmed. When composing error messages, developers should focus more on the message completeness, especially on whether crucial runtime contexts are being included. In addition, instead of stating what is *not* allowed, error messages should articulate what *is* allowed whenever possible. Furthermore, error messages that are clear to developers might not be perceived the same way by end users. Developers should be more aware of this conceptual mismatch in order to design user-friendly error messages.

For researchers, our study brings a few new perspectives on error message quality and fixing strategies. First, we observed that for a great number of error messages, their "deficiency" was inherently hard to resolve due to unknown user intentions or indiscretions. This finding urges related research to differentiate error messages more carefully based on their specific deficiency causes. Second, while most related research emphasized the problematic content of error messages (Section 7.1), we observed that content improvements in fact account for only a small proportion of error-message-related fixes. On the other hand, most error message issues were fixed by employing better exception handlings. However, although exception handling has been actively studied, very few work pointed out its impact on error message quality (Section 7.2). Our work identifies an important intersection of these two lines of work, hence better motivates future research on related topics.

6 THREATS TO VALIDITY

Our study analyzed a subset of error messages from six data science libraries. The results may not generalize to error messages from other software or application domains. We mitigate this threat by maximizing the representativeness of the subject libraries. As described in Section 2.1, the subjects are widely used in various data science contexts.

As our subjects are all Python libraries, certain observations might be specific to the inherent characteristics of dynamic programming languages (e.g., error messages related to type errors). Nevertheless, most of our qualitative findings are language-agnostic. For example, deficient error messages that are caused by ambiguous user intentions or cognitive gaps should also exist for statically-typed languages.

We used Stack Overflow and GitHub as the primary data sources, which could have noises. To mitigate this threat, we used only trustworthy information, such as the upvoted SO questions, accepted SO answers, and merged commits, for the analyses.

We derived the deficiency causes and fixing strategies of error messages mainly from manual inspections, which could be subjective. To mitigate this threat, we adopt the inductive coding process with multiple human annotators, so that the qualitative results were derived iteratively and collaboratively (Section 2). We also released the results [33] so that researchers could help further reduce the threat.

We characterized error messages in terms of whether they pinpoint the fault (i.e., fault localization) and how they deliver the resolutions (i.e., repairs). Admittedly, there could be other ways to evaluate error message quality. We chose these two metrics since fault localization and program repair are both important milestones in debugging [15].

7 RELATED WORK

7.1 Error message quality

Barik et al. found through an eye tracking study that developers allocate substantial debugging efforts on understanding error messages, which are equally challenging as understanding source code [4]. Unfortunately, deficient error messages have been observed in different programming and application domains. For example, Zhang and Ernst proposed a technique to detect inadequate diagnostic messages for software configuration errors [39]. Xu et al. identified the lack of feedback information in access-denied log messsages, which imposed unnecessary obstacles for system administrators [36]. Marceau et al. studied novices' interactions with the error messages generated by an IDE for the Racket programming language [24]. Barik et al. applied *Toulmin's model of argument* to study the design principles of Java compiler error messages [3]. Thiselton and Treude proposed a tool that automatically queries Stack Overflow and summarizes answers to enhance Python compiler error messages [34].

Nonetheless, few studies explicitly distinguished the types of deficient error messages and analyzed the underlying reasons. On the contrary, our work quantified the diagnostic informativeness of error messages in practical debugging context, and found that error messages are often inevitably misleading or uninformative. We also observed that the majority of error-message-related issues were fixed by updating source code logics, which seems to be overlooked by prior research that focused exclusively on improving error message content. Finally, our study complements the previous work by investigating the quality of error messages in the emerging domain of data science.

7.2 Exception handling

Exception handling have been actively studied by researchers. For example, Sena et al. empirically studied the exception flow and handler actions of 656 Java libraries to understand the exception handling strategies [31]. Kechagia et al. investigated the exception handling mechanisms of Android APIs [21]. Other work focused on exception handling related bugs. For example, Ebert et al. proposed a classification of exception handling bugs, based on a survey and an analysis of 220 Java exception handling bugs [7]. Coelho et al. mined exception stack traces from Android issues to investigate bug hazards related to exception handling code [6]. A few studies proposed techniques to automatically generate exception handling code. A recent work from Nguyen et al. proposed to learn fuzzy logic rules from high-quality programs to recommend exception handling code [27]. Zhang et al. proposed a neural approach that predicts the locations of try blocks and automatically completes catch blocks [37].

Nonetheless, most of these studies highlight program crashes or resource leakage as the consequences of error handling bugs. Our work pointed out that unthoughtful error handlings could also lead to confusing, obscure, and unhelpful error messages that seriously impede debugging.

7.3 Debugging data science programs

Debugging data science programs has received much research attention in recent years. Zhang et al. empirically studied the root causes, symptoms, and detection strategies of TensorFlow program bugs [40]. Islam et al. investigated bug characteristics [18] and fix patterns [19] of popular deep learning libraries including Caffe, Keras, TensorFlow, Theano, and Torch. Zhang et al. analyzed real failures from a deep learning platform in Microsoft and conducted developer interviews to understand debugging practice on deep learning jobs [38]. In addition to empirical studies, researchers have also proposed techniques to automatically debug and test data science programs. For example, Pham et al. proposed CRADLE, which detects bugs in deep learning libraries by leveraging cross-implementation inconsistency checking and anomaly propagation tracking [29]. Zhang et al. proposed a static analysis approach, DEBAR, to detect numerical bugs in neural network architectures [41]. Our study complements the previous work by examining the diagnostic capabilities of error messages in different debugging contexts of data science programs.

8 CONCLUSION

In this work, we empirically demystified the causes and fixes of “bad” error messages in data science libraries. We found that the “misleadingness” and “uninformativeness” of error messages are often inevitable and hard to fix, since libraries are inherently unaware of user intentions and errors. We also found that “bad error messages” are not equivalent to “bad phrasing of message content”, given that source-code and program-logic changes are often required to fix error-message-related complaints. We hope that these findings, which significantly deepen the understanding of the origins, perceptions, and coping strategies of bad error messages, could effectively steer future research in this field.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their constructive suggestions and insightful comments. This work was partially supported by the National Natural Science Foundation of China under Grants No. 61836005, 61772347, 61972260, 61872273, Guangdong Basic and Applied Basic Research Foundation under Grant No. 2019A1515011577, and Guangdong Provincial Key Laboratory under Grant No. 2020B121201001.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] Titus Barik. 2016. How should static analysis tools explain anomalies to developers?. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 1118–1120. https://doi.org/10.1145/2950290.2983968

[3] Titus Barik, Denae Ford, Emerson Murphy-Hill, and Chris Parnin. 2018. How should compilers explain problems to developers?. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 633–643. https://doi.org/10.1145/3236024.3236040

[4] Titus Barik, Justin Smith, Kevin Lubick, Elisabeth Holmes, Jing Feng, Emerson Murphy-Hill, and Chris Parnin. 2017. Do developers read compiler error messages?. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 575–585. https://doi.org/10.1109/ICSE.2017.59

[5] Yanto Chandra and Liang Shang. 2019. *Qualitative research using R: A systematic approach*. Springer.

[6] Roberta Coelho, Lucas Almeida, Georgios Gousios, and Arie Van Deursen. 2015. Unveiling exception handling bug hazards in Android based on GitHub and Google code issues. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. 134–145. https://doi.org/10.1109/MSR.2015.20

[7] Felipe Ebert, Fernando Castor, and Alexander Serebrenik. 2015. An exploratory study on exception handling bugs in Java programs. *Journal of Systems and Software* 106 (2015), 82–101. https://doi.org/10.1016/j.jss.2015.04.066

[8] Charles R. Harris et al. 2020. Array programming with NumPy. *Nature* 585 (2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

[9] Pauli Virtanen et al. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2

[10] Stack Exchange. 2021. How to Create a Minimal, Reproducible Example. https://stackoverflow.com/help/minimal-reproducible-example

[11] Stack Exchange. 2021. Stack Exchange API. https://api.stackexchange.com/

[12] Python Software Foundation. 2021. Python Concepts. https://docs.python.org/3/glossary.html.

[13] Python Software Foundation. 2021. Python Input and Output. https://docs.python.org/3/tutorial/inputoutput.html.

[14] Python Software Foundation. 2021. Python inspect - Inspect live objects. https://docs.python.org/3/library/inspect.html.

[15] L. Gazzola, D. Micucci, and L. Mariani. 2019. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* 45, 1 (2019), 34–67. https://doi.org/10.1109/TSE.2017.2755013

[16] GitHub. 2021. GitHub REST API. https://docs.github.com/en/free-pro-team@latest/rest

[17] Jingmei Hu, Jiwon Joung, Maia Jacobs, Krzysztof Z Gajos, and Margo I Seltzer. 2020. Improving data scientist efficiency with provenance. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1086–1097. https://doi.org/10.1145/3377811.3380366

[18] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. 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)*. 510–520. https://doi.org/10.1145/3338906.3338955

[19] Md Johirul Islam, Rangeet Pan, Giang Nguyen, and Hridesh Rajan. 2020. Repairing deep neural networks: Fix patterns and challenges. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 1135–1146. https://doi.org/10.1145/3377811.3380378

[20] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *2013 35th International Conference on Software Engineering (ICSE)*. 672–681.

[21] Maria Kechagia, Marios Fragkoulis, Panos Louridas, and Diomidis Spinellis. 2018. The exception handling riddle: An empirical study on the Android API. *Journal of Systems and Software* 142 (2018), 248–270. https://doi.org/10.1016/j.jss.2018.04.034

[22] Miryung Kim, Thomas Zimmermann, Robert DeLine, and Andrew Begel. 2017. Data scientists in software teams: State of the art and challenges. *IEEE Transactions on Software Engineering* 44, 11 (2017), 1024–1038. https://doi.org/10.1109/TSE.2017.2754374

[23] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. 2014. The Stanford CoreNLP Natural Language Processing Toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*. 55–60. https://doi.org/10.3115/v1/P14-5010

[24] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011. Mind your language: on novices' interactions with error messages. In *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*. 3–18. https://doi.org/10.1145/2048237.2048241

[25] M. McHugh. 2012. Interrater reliability: the kappa statistic. *Biochemia Medica* 22 (2012), 276 – 282.

[26] Wes McKinney. 2010. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference (SciPy'10)*, Vol. 445. 51–56. https://doi.org/10.25080/Majora-92bf1922-00a

[27] Tam Nguyen, Phong Vu, and Tung Nguyen. 2020. Code recommendation for exception handling. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1027–1038. https://doi.org/10.1145/3368089.3409690

[28] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[29] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. 2019. CRADLE: cross-backend validation to detect and localize bugs in deep learning libraries. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 1027–1038. https://doi.org/10.1109/ICSE.2019.00107

[30] Radim Řehůřek and Petr Sojka. 2010. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. ELRA, 45–50.

[31] Demóstenes Sena, Roberta Coelho, Uirá Kulesza, and Rodrigo Bonifácio. 2016. Understanding the exception handling strategies of Java libraries: An empirical study. In *Proceedings of the 13th International Conference on Mining Software Repositories*. 212–222. https://doi.org/10.1145/2901739.2901757

[32] Jim Shore. 2004. Fail fast [software debugging]. *IEEE Software* 21, 5 (2004), 21–25. https://doi.org/10.1109/MS.2004.1331296

[33] Yida Tao, Zhihui Chen, Yepang Liu, Jifeng Xuan, Zhiwu Xu, and Shengchao Qin. 2021. *Artifacts for "Demystifying "Bad" Error Messages in Data Science Libraries"*. https://doi.org/10.5281/zenodo.4889055

[34] Emillie Thiselton and Christoph Treude. 2019. Enhancing Python Compiler Error Messages via Stack Overflow. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 1–12.

[35] Vicente Javier Traver Roig. 2010. On compiler error messages: what they say and what they mean. *Adv. in Hum.-Comp. Int.* (2010). https://doi.org/10.1155/2010/602570

[36] Tianyin Xu, Han Min Naing, Le Lu, and Yuanyuan Zhou. 2017. How do system administrators resolve access-denied issues in the real world?. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. 348–361. https://doi.org/10.1145/3025453.3025999

[37] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Yanjun Pu, and Xudong Liu. 2020. Learning to Handle Exceptions. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 29–41.

[38] Ru Zhang, Wencong Xiao, Hongyu Zhang, Yu Liu, Haoxiang Lin, and Mao Yang. 2020. An empirical study on program failures of deep learning jobs. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 1159–1170. https://doi.org/10.1145/3377811.3380362

[39] Sai Zhang and Michael D Ernst. 2015. Proactive detection of inadequate diagnostic messages for software configuration errors. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 12–23. https://doi.org/10.1145/2771783.2771817

[40] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An empirical study on TensorFlow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 129–140. https://doi.org/10.1145/3213846.3213866

[41] Yuhao Zhang, Luyao Ren, Liqian Chen, Yingfei Xiong, Shing-Chi Cheung, and Tao Xie. 2020. Detecting numerical bugs in neural network architectures. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 826–837. https://doi.org/10.1145/3368089.3409720