An Empirical Study of API Misuses of Data-Centric Libraries

Akalanka Galappaththi
akalanka@ualberta.ca
University of Alberta
Edmonton, Canada

Sarah Nadi
sarah.nadi@nyu.edu
New York University Abu Dhabi
Abu Dhabi, United Arab Emirates

Christoph Treude
ctreude@smu.edu.sg
Singapore Management University
Singapore

ABSTRACT

Developers rely on third-party library Application Programming Interfaces (APIs) when developing software. However, libraries typically come with assumptions and API usage constraints, whose violation results in *API misuse*. API misuses may result in crashes or incorrect behavior. Even though API misuse is a well-studied area, a recent study of API misuse of deep learning libraries showed that the nature of these misuses and their symptoms are different from misuses of traditional libraries, and as a result highlighted potential shortcomings of current misuse detection tools. We speculate that these observations may not be limited to deep learning API misuses but may stem from the data-centric nature of these APIs. Data-centric libraries often deal with diverse data structures, intricate processing workflows, and a multitude of parameters, which can make them inherently more challenging to use correctly. Therefore, understanding the potential misuses of these libraries is important to avoid unexpected application behavior. To this end, this paper contributes an empirical study of API misuses of five data-centric libraries that cover areas such as data processing, numerical computation, machine learning, and visualization. We identify misuses of these libraries by analyzing data from both Stack Overflow and GitHub. Our results show that many of the characteristics of API misuses observed for deep learning libraries extend to misuses of the data-centric library APIs we study. We also find that developers tend to misuse APIs from data-centric libraries, regardless of whether the API directive appears in the documentation. Overall, our work exposes the challenges of API misuse in data-centric libraries, rather than only focusing on deep learning libraries. Our collected misuses and their characterization lay groundwork for future research to help reduce misuses of these libraries.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**.

KEYWORDS

API misuse, data-centric libraries, empirical study

ACM Reference Format:
Akalanka Galappaththi, Sarah Nadi, and Christoph Treude. 2024. An Empirical Study of API Misuses of Data-Centric Libraries. In *Proceedings of the 18th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '24), October 24–25, 2024, Barcelona, Spain.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3674805.3686685

ESEM '24, October 24–25, 2024, Barcelona, Spain

ACM ISBN 979-8-4007-1047-6/24/10
https://doi.org/10.1145/3674805.3686685

1 INTRODUCTION

When developing software applications, developers typically use third-party libraries that offer access to various functionality through a set of *Application Programming Interfaces* (APIs). While some APIs are easy to use and integrate, others have certain usage constraints that application developers need to follow in order to correctly achieve the desired functionality. Violating these constraints leads to incorrect API usage, also referred to as an *API misuse* [3, 48, 59]. API misuses can lead to program crashes, security vulnerabilities, performance problems, or unexpected program behavior [48].

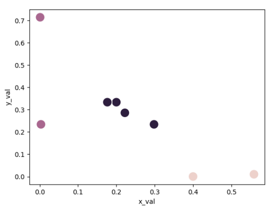
There is a long line of research studying API misuses [2, 21, 22, 36, 45, 54, 56, 58, 59], most of which focused on Java APIs and typically considered misuses related to the control flow and data flow between APIs [2, 48]. For example, a common API misuse is missing `null` checks or forgetting certain API calls (e.g., not calling `close()` after calling `open()` on a stream).

In recent work, Wei et al. [57] investigated API misuses of two Python deep learning libraries, TensorFlow and PyTorch. They found that many of the misuses are caused by incorrect device usage (e.g., CPU instead of GPU) or under-the-hood data conversion problems. They conclude that misuses like data shape mismatch in deep learning libraries are unique, because they are harder to detect, and also because they may not immediately raise errors but instead produce incorrect results or lead to performance problems. However, we observe that apart from CPU/GPU misuses, many of their observed misuses may not be specific to deep learning per se but may rather stem from the reliance on data. We also observe that there are many other libraries that focus on processing, analyzing, and deriving insights from data. For example, data processing and manipulation libraries such as pandas [40] or visualization libraries such as seaborn [52] all deal with diverse data structures, intricate processing workflows, and a multitude of parameters, which can make them inherently more challenging to use correctly. We use the term *data-centric libraries* to refer to such libraries. Given their shared focus on handling data, it is reasonable to speculate that some of the new types of API misuses observed for deep learning libraries may also manifest in other data-centric libraries.

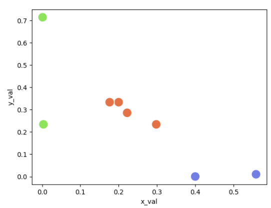
Consider the following deep learning API misuse example provided by Wei et al. [57]: a developer intends to multiply two tensors, A and B, where A is a 3×3 tensor and B is a 2×3 tensor. Since the dimensions are not compatible for multiplication, the developer needs to first transform tensor B. In TensorFlow, when transforming a 2×3 tensor to a 3×2 tensor, the developer has two options: `reshape` or `transpose`. While both methods produce tensors with similar shapes, `transpose` is the correct API to call in this

```
1  import pandas as pd
2  import matplotlib.pyplot as plt
3  import seaborn as sns
4
5  # df dataframe has format:
6  #     x_values  y_values  color
7  # 0  0.200000  0.333333    2.0
8  # 1  0.003334  0.234043    1.0
9  # ..  ..        ..        ..
10 # 7  0.003334  0.234043    1.0
11
12 df = ...
13
14            #violet     #green     #orange
15 colors = ['#747FE3', '#8EE35D', '#E37346']
16 - sns.set_palette(colors)
17
18 - sns.scatterplot(data = df, x="x_values", y="y_values", hue="color", s=40,
         legend=False)
19 + sns.scatterplot(data = df, x="x_values", y="y_values", hue="color", s=40,
         legend=False, palette=colors)
20 plt.show()
```

(a) Seaborn API misuse of `set_palette` and `scatterplot`



(b) Incorrect hue colors



(c) Correct hue colors

Figure 1: Example of seaborn API misuse and its impact, based on Stack Overflow question 67637829.

scenario, because with `transpose`, the dimensions are appropriately matched ($[3 \times 3] \times [3 \times 2] \rightarrow [3 \times 2]$) and elements are correctly positioned. While using `reshape` may not immediately appear incorrect since it returns expected dimensions, using this API leads to incorrect output as the elements of B are rearranged improperly. Importantly, this misuse does not trigger an error but rather yields an incorrect output. Any subsequent operations using the resulting tensors are also compromised.

Now, let us consider the scenario shown in Figure 1a where a developer uses the Python visualization library seaborn to create a scatterplot. In this example, the developer wants to color points on the scatterplot differently according to the value in the column `color` of their data frame, which is why they set `hue="color"` on Line 18. The variable `colors` on Line 15 contains the specific colors they want to use. Seaborn offers different options to set color palettes such as calling the `set_palette(...)` method or passing a value to the `palette` parameter in the `scatterplot` API (e.g., `scatterplot(..., palette=...)`). On Line 16, they use `set_palette` to assign these colors as the palette to use. However, the displayed scatter plot in Figure 1b colors the points using a different palette than the one specified; the correctly colored scatterplot is shown in Figure 1c. It turns out that the use of the specified color palette depends on the type of data in the column used as the `hue`. In cases of numerical values, seaborn defaults to an internal color palette, completely ignoring whatever value was set using `set_palette`. Had the type of the `color` column been categorical (i.e., strings) such as "zero", "one", "two", the same code would have correctly used the specified colors. The only way to get the desired behavior with a numerical column is to pass `palette=colors` as an argument to `scatterplot` on Line 19 instead of calling `set_palette` on Line 16. Thus, the data inside the data frame affects the correctness of the API usage, highlighting the importance of understanding how data types influence the behavior of the library.

These examples demonstrate that some of the challenges and unique nature of API misuses found for the two deep learning libraries Tensorflow and PyTorch [5, 23, 57] may extend to other commonly used Python libraries, specifically those with a data-centric nature. In general, misuses caused by data conversion errors are not necessarily limited to deep learning or even machine learning libraries. Our goal in this work is to more broadly investigate API misuses of data-centric libraries.

Specifically, this paper presents an empirical study of API misuse in data-centric libraries. We focus on Python, the most popular language for data-driven applications [6, 16]. Since deep learning libraries have been already studied [57], we selected five additional widely-used data-centric Python libraries: NumPy [37], pandas [40], scikit-learn [49], Matplotlib [27], and seaborn [52].

To identify misuses, we manually analyze 345 Stack Overflow posts and 358 commits from open-source GitHub repositories that use these libraries. Using information from these two different data sources allows us to discover both API misuses that make their way to developer's committed code, as well as those where developers seek community help before finalizing their code. We then categorize the identified misuses in terms of the misuse type, exhibited symptom, and root cause using Wei et al.'s taxonomy [57], which is derived from the study of misuses of deep learning libraries. We additionally investigate whether the library documentation includes any explicit guidelines for avoiding the identified misuses.

Overall, we collect 49 API misuses, covering 45 distinct APIs. Our findings reveal that despite differences in programming paradigms between deep learning and traditional machine learning [57] or visualization libraries, the nature of misuses previously observed extend beyond the deep learning libraries. Specifically, we find that 39% of the misuses are due to data-conversion errors and the most misused element is API parameters (51%). However, we find the need to extend the existing taxonomy with an additional dimension of *data dependency* to capture the idea that the exact same API usage may be a misuse in one case while a correct usage in another, only depending on the data being processed. The example discussed in Figure 1a is an example of a data-dependent misuse. We find that 55% of the studied misuses are data dependent and 35% of misuses result in incorrect output without any explicit run-time errors, while 41% cause program crashes. Overall, we find that the common characteristic between all these libraries that leads to the API misuses is their heavy reliance on data. Surprisingly, we find that 39% of the misuses have documented API directives, but developers still misused the API.

Our findings have implications for language and API design, misuse detectors, and surfacing information buried in documentation. Specifically, data-centric libraries need to make assumptions about data content and format and yet there are no built-in language mechanisms for helping API designers enforce these assumptions. Even when they document these assumptions, client developers end up misusing them, further implying that we need more research on surfacing important information in documentation.

In summary, this paper makes the following contributions:

- We define the notion of data-centric libraries, and show that API misuses of those libraries share characteristics with previously identified API misuses of deep learning libraries.
- We collect 49 API misuses from 5 Python libraries, corresponding to misuses of 45 distinct APIs.
- We categorize the detected misuses using an existing deep-learning misuse taxonomy, finding that it is expressive enough to categorize the majority of the data-centric misuses. However, we had to introduce the notion of data dependency to the taxonomy to capture some of our observations.
- We identify if the misused APIs have corresponding guidelines in the library documentation.
- For each misuse, we construct a reproducible example to illustrate the misuse.
- We discuss the implications of our findings on API and language design, surfacing information buried in documentation, and designing misuse detectors.

Our data is available on our online artifact page [4].

2 BACKGROUND, SCOPE, AND DEFINITIONS

2.1 Definitions and Scope

Schlichtig et al. [48] noted some discrepancies in how researchers define API misuse. For example, some authors consider only the external API of a library [3, 20], while others consider also internally defined APIs that are meant to be used only inside the current application [11]. Similarly, some work considers client code that uses an outdated API due to breaking changes in the library as a misuse [11, 48, 57], while other work considers general Python typing issues as misuses [11]. In this work, we are interested in API misuses of third-party libraries that stem from the unique nature of the data-centric domain, rather than the nature of the underlying programming language or general software evolution characteristics. To clearly define our scope, we use the following definitions from Schlichtig et al. [48], with some adaptations, if needed, shown in non-italic square parentheses.

Definition I. An ***Application Programming Interface*** *(API)* is the *"public interface* [that] *exposes software elements (e.g., classes and methods) to the outside world, making the implemented functionality accessible [48]."* We focus only on the use of third-party library APIs in client code, considering all public API elements such as classes, methods (including their parameters), and attributes.

Definition II. "*An* ***API directive*** *is a natural-language statement related to guidelines or constraints that describes how to use an API correctly and optimally* [, regardless of the developer's task or intention]. *It can be part of the underlying documentation of an API. However, an API directive can also be implicit, for example, because of incomplete documentation or expected domain-specific knowledge* [48]."

Definition III. "*An* ***API usage constraint*** *is an API directive that restricts the actual use of an API. These restrictions are not enforced by the programming language itself, such as correct typing* [, nor are they part of the natural software evolution process (e.g., deprecation)]. *Because API usage constraints are API directives, they are imposed by the API designer/expert* [48]."

For example, the `scatterplot` documentation contains this API directive, related to the misuse in Figure 1: *"The default treatment of the hue (and to a lesser extent, size) semantic, if present, depends on whether the variable is inferred to represent "numeric" or "categorical" data. In particular, numeric variables are represented with a sequential colormap by default, [...] This behavior can be controlled through various parameters, as described [...] below. [51]"*.

Definition IV. "*An* ***API misuse*** *is the violation of one or more API usage constraints. Such violation leads to misbehaviour of the API, e.g. errors, crashes,* [incorrect output,] *or vulnerabilities* [48]."

Note that the API misuse definition depends on the definition of an API usage constraint. The distinction in Definition IV is important for our work, since we do not want to consider errors that are inherent to programming in Python regardless of the library being used. For example, if a developer passes a float argument to an integer parameter, the Python type system will raise a run-time TypeError *integer argument expected, got float.* We do *not* consider this as an API misuse since it is a typical programming error that stems from the dynamic typing nature of Python rather than violating an API usage constraint. This is, for example, a key distinction between our definition of API misuse (as well as Wei et al. [57]'s) and that of He et al. [11]'s study of general Python API misuses.

Our explicit exclusion of deprecations from Definition IV is also important. Deprecations and breaking changes are a general phenomenon for all dependencies/libraries, regardless or the programming language or nature of these libraries, and there are typically warnings in place for such deprecations. Thus, we do not consider modifications to an API usage that occur due to compliance with future library changes, as such adaptations are inevitable with the evolution of libraries. While Wei et al.'s study [57] considered deprecation management errors as misuses, accounting for approximately 20% of the collected misuses, deprecations are not specific to deep learning libraries in any way. Our focus is on identifying misuses in the domain of data-centric libraries, while deprecations would be observed for any library from any domain.

Note the slight variation we added in Definition III of API directive, w.r.t. the developer's intention. For example, a developer accidentally passing "write" as the open mode of a file instead of "append" is not a misuse since both these modes are correct and depend on what the developer wants to do. We cannot assume the developer's intention of whether they want to write or append to the file. A misuse should always be a misuse, without requiring extra information about the developer's intention. In contrast, setting the axis color of a plot without enabling the axis first is a misuse since any code that sets the axis color without enabling the axis is problematic (the axis color would not have any effect).

Finally, we note one interesting distinction by Schlichtig et al. [48] in the following definition:

Definition V. *"A* ***domain-specific API*** *offers functionality tailored to a specific domain. Its domain determines the achievable goal and application rules of a domain-specific API. In contrast,* ***non-specific APIs*** *are not tailored to a domain, nor do they determine a specific goal associated with their use [48]."*

Schlichtig et al. [48] argued that this distinction is important because domain-specific APIs sometimes have additional or very specific types of usage constraints, as exhibited by API misuse in the cryptography domain for example [17, 34]. Data-centric libraries offer domain-specific APIs, and previous studies of deep learning libraries already showed that characteristics such as accounting for data shape in these libraries caused new types of misuses, posing challenges to traditional misuse detection tools. Both of these facts motivate the study in this paper, where we extend the scope beyond only deep learning libraries to understand API misuses in the general domain of data-centric libraries.

2.2 API Misuse of Deep Learning Libraries

Our study is motivated by the findings of Wei et al. [57] who examined API misuse of Python deep learning libraries, particularly TensorFlow and PyTorch. To identify misuses, they mined the change history of various client repositories of these two libraries and manually analyzed candidate commits that have made changes to the libraries' API usages. Based on the identified misuses, they create a taxonomy of API misuse types, root causes, and impact. While this taxonomy amalgamates information from existing work [3, 11, 20, 48], the authors add new root causes that stem from the nature of deep learning libraries, specifically data conversion problems and device management errors. Figure 2 shows the summary of their developed taxonomy, which we leverage for labeling the misuses we discover in our work. The taxonomy first categorizes the types of misuses by the program element involved in the misuse as well as the type of the violation (e.g., misuse in Figure 1 would be a missing API parameter). API method refers to misuses related to calling functions or methods. API parameter are the parameters of the API methods. API condition refers to conditional statements required for the API usage. The taxonomy also categorizes the root cause of the problem. Devise management errors relate to any hardware or resource utilization (e.g., using CPU vs GPU). Algorithm errors are related to mathematical problems (e.g., dividing by zero) or incorrect calculations. Data conversion errors represent issues that relate to incorrect conversions between data types or shapes. When a program accesses a null object, null reference errors occur. Deprecation management errors relate to the usage of deprecated APIs or parameters. Finally, the taxonomy categorizes the observed symptom, which in our example would be unexpected output. Program crashes and warnings refer to runtime errors and warnings. Low efficiency refers to slow program execution.

Note that the figure shows some of our modifications to the taxonomy in terms of adding items (in red, discussed later in our results), or not considering items due to our goal and defined scope above. For the latter, we do not consider the greyed out outdated violation and root cause of deprecation management errors.

3 METHODS

We conduct an empirical study to identify API misuses in data-centric libraries. Most previous research on API misuse identified misuses from either Stack Overflow or git history on GitHub [13, 15, 60]. In our work, we use both data sources to collect API misuses. We manually analyze Stack Overflow questions and their answers to determine if they are discussing a problem related to API misuse. We also investigate commits from Python projects on GitHub, as they may contain fixes for API misuses. For confirmed misuses from either data source, we note how the misuse happened (e.g., passing incorrect argument type) and what was the observed symptom (e.g., incorrect output) by reproducing the misuse. We then use Wei et al. [57]'s misuse classification of deep learning API misuses to categorize the misuse types, their root causes, and symptoms. As a new contribution of our work, we consult the library's documentation to determine if there is a documented API directive related to the misuse.

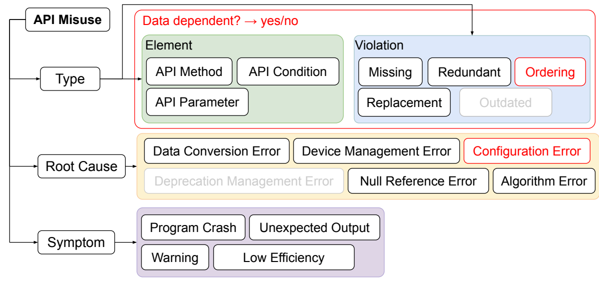


Figure 2: Our updated misuse classification taxonomy, based on Wei et al. [57]'s analysis of deep learning misuses. Greyed out boxes are out-of-scope items we do not consider, while red boxes show our additions.

3.1 Data Collection

3.1.1 Libraries. We select five popular data-centric Python libraries [44]: NumPy [37], pandas [40], scikit-learn [49], Matplotlib [27], and seaborn [52]. NumPy is a fundamental package for scientific computing with Python, providing support for large, multidimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays. Pandas is a fast, powerful, flexible, and easy-to-use data analysis and data manipulation library. Scikit-learn is a popular library for data preprocessing, machine learning, and visualization. Matplotlib and seaborn are comprehensive libraries for creating interactive visualizations in Python. We consider both Matplotlib and seaborn as they are both commonly used together for visualizations.

3.1.2 Selecting questions in Stack Overflow. Stack Overflow [39] is a widely used question and answer website that is popular among developers and programming enthusiasts. When posting questions, users are encouraged to provide minimal reproducible code examples, stack trace of errors if any, and problem descriptions. Stack Overflow has tags that represent various technologies, which can highlight the libraries discussed in questions. We utilize these characteristics in Stack Overflow questions to identify questions that potentially relate to API misuses.

Since Stack Overflow has tags to represent various libraries, we retrieve Stack Overflow posts tagged with each of our libraries using the Stack Exchange data explorer [38]. To increase our chances of finding questions related to API misuse, we use the following additional filtering criteria. We remove questions with no answers since answers are essential to help us determine the root cause behind the posed problem. We filter out questions whose title starts with "how to" since these relate to the poster asking about how to implement particular functionality, which is unlikely to relate to API misuse. We also filter out questions with negative scores, because a negative score indicates poorly articulated questions or those missing information [35]. Additionally, we only collect the questions that were posted between 2019 and 2023 in order to avoid analyzing outdated questions. Finally, we filter out questions with

Table 1: Number of collected and filtered Stack Overflow questions for each library, collected on 31-05-2023.

Tag	Total # of questions	# of questions after filtering	Minimum sample size	Annotated sample size
pandas	106,331	95,992	69	69
NumPy	30,698	27,774	68	69
scikit-learn	7,132	6121	68	69
Matplotlib	16,916	15,463	68	69
seaborn	3,251	2,942	67	69
Total	164,328	148,116	340	345

no code snippet in the question body, as we need to analyze a code snippet to determine whether it contains an API misuse.

Table 1 shows the total number of questions in Stack Overflow tagged with each library and the number of questions that we collect after applying the above filtering criteria. Since manually analyzing close to 164,000 posts is infeasible, we resort to analyzing a sample of these threads. We find that the question scores exhibit a right-skewed distribution where the majority of the questions either have a score of zero, one, or two while only a few questions have a score higher than two. Accordingly, we partition the questions into three groups based on score: zero, one or two, and three or higher and then we randomly select questions from each group to annotate (i.e., we perform stratified sampling according to question score). We choose 90% confidence level and 10% error margin when selecting samples from each question group (i.e. tag). The resulting sample sizes varied between 67-69 per tag, which provided us with a target of 340 questions to be analyzed. For the sake of dividing questions equally among three annotators, we analyze 345 Stack Overflow questions in total.

3.1.3 Selecting commits in GitHub projects. Our goal is to find Python projects that use our libraries of interest and analyze their commit history to identify potential fixes of API misuses of these libraries. For each analyzed library, we use the GitHub dependency graph of the library's public repository to find client projects. After collecting all dependents of the five libraries, we remove the duplicates as some of the dependent projects use more than one library (e.g., pandas and NumPy). After removing duplicates, we find 869,544 projects that use at least one of the five libraries. We collect the metadata of all these dependent repositories; specifically, stargazer count, number of contributors, programming language, visibility (public or private), archived or not, fork or not, project creation date, and the last committed date. From the collected metadata, we remove projects whose main programming language is not Python. We filter out projects that are either private or archived. We also filter out forks. After removing repositories using the above filters, we sort the remaining repositories in descending order based on stargazer count, number of contributors, and the project's age. We calculate the age by subtracting the created date from the last committed date. We then select the top 100 projects from this ranked list to analyze their commits. From the selected 100 repositories, we select candidate commits that may contain API misuses for manual evaluation. Specifically, a commit that contains changes to an API usage of one the five libraries could potentially be a fix for a misuse.

To find candidate commits, we use PyDriller [42] to first detect the modified Python files in each commit along with the modified line numbers. We ignore changed files with less than 10 changed lines, as previous studies observed that fixing API misuses typically involved small number of edits [3, 57]. We also filter out commits that are likely fixing a typo than changing the API usage by using a regular expression to determine if the commit message is a one-liner, and the message contains "fix" or "correct" with "typo". For the remaining commits, we parse the changed Python files to find if any of the modified lines contain API usages of the target libraries.

Table 2: Number of candidate commits for each library.

Library	# of all commits	# of diverse commits	# of reviewed commits	# of unique APIs	# of reviewed APIs
pandas	390	185	50 (27%)	92	37 (40%)
NumPy	2273	431	103 (24%)	184	74 (40%)
scikit-learn	545	528	143 (11%)	210	84 (40%)
Matplotlib	234	146	57 (39%)	80	42 (53%)
seaborn	5	5	5 (100%)	8	8 (100%)
Total	3447	1,295	358 (28%)	574	245 (43%)

Overall, we find 3,447 candidate commits originating from 76 of the analyzed projects. Table 2 shows the number of candidate commits per library. In these commits, we find that APIs such as `numpy.array` or `pandas.DataFrame` were modified over hundred times while APIs such as `seaborn.heatmap` or `matplotlib.Axes3D` were modified less than ten times. Additionally, many APIs were modified only once. To prioritize the discovery of unique API misuses, we focus on selecting commits that modify different APIs. For APIs that appear in less than three candidate commits, we include all related commits for those changes. However, if an API appears in more than three candidate commits, we randomly select three commits from different projects. From the 3,447 candidate commits, we first select commits that modify unique APIs, resulting in identifying 1,295 diverse commits (Table 2 column 3). These commits contain 574 unique APIs that were modified. We manually analyze 358 randomly selected diverse commits (Table 2 column 4) to identify API misuses, satisfying a 95% confidence interval and 5% error margin.

3.2 Data Analysis

We rely on manually analyzing a given candidate Stack Overflow post or GitHub commit to determine if it contains an API misuse. We create a coding guideline to streamline this process. In the subsections below, we first describe how we iteratively developed our coding guideline using Stack Overflow data and then describe the developed criteria. Finally, we discuss our closed coding process for labeling the identified misuses.

3.2.1 Developing a coding guideline: Stack Overflow threads typically have a wealth of information, including problem descriptions, code examples, stack traces, comments, and answers. Thus, we use the process of analyzing Stack Overflow posts to iteratively develop and refine our coding guide for confirming API misuse.

Specifically, we first conduct the manual annotation process of Stack Overflow posts iteratively using three authors of this paper. We start with an initial coding guide on how to identify API misuses. After each iteration, annotators discuss any disagreements that occur in that round. If a resolution surfaces any new information that has not been included in the initial coding guide, we improve the coding guideline to be more explicit. In each round of the first five rounds, we select six Stack Overflow questions for each library, totaling 30 questions per round. After each round, we calculate the agreement score for determining a misuse using Fleiss kappa score [8]. In the first three rounds, we reach moderate agreement (>0.5) [18]. In round four, our agreement was fair (0.37) [18]. After revising the coding guide, in the fifth round, we received an almost perfect agreement score (0.85) [18] which indicates that the annotators achieved a solid understanding on how to identify and distinguish API misuses and that the instructions in the code guide are stable. After the fifth round, the same three authors continue analyzing the remaining threads but with only two annotators per thread. Overall, we maintain a moderate agreement (>0.5), measured through a pair-wise kappa score [18], during the last round of annotation. Each pair of annotators resolved their disagreements, sometimes involving the third annotator if needed. Given the stability of the coding guideline at this point, we proceed to assign only one annotator per selected GitHub commit. We then perform an additional verification step where we create a minimal reproducible example for each identified misuse to confirm that our understanding of the misuse is correct.

3.2.2 Coding guideline & coding process. Our exact coding guideline is part of our artifact. When analyzing a Stack Overflow thread, we make use of the question title, description, provided code snippets, as well as the answers in the thread. We discard Stack Overflow posts that mainly seek help with implementing a particular functionality (e.g., *best way to plot two graphs on the same axis*) or understanding code/concepts (e.g., *why does a specific function use a particular approximation method*). On the other hand, when analyzing GitHub commits, we make use of the commit message to understand the purpose of the change, as well as reading through the modified code lines. In some commits where a bug issue or pull request is linked in the commit message, we also read through the conversation there to get more context about the fix. We now describe some of the important criteria for determining a misuse.

If the problem is related to the user's intention (e.g., they want the legend on the right not the left), we do not consider this as a misuse (Definition IV). Additionally, if the fix is related to managing API deprecation, we do not consider this as a misuse (Definition III). To differentiate API misuses from normal Python (type) errors, we do not consider problems that are caught by the Python type system as a misuse (e.g., passing float instead of integer).

Once we determine that a thread/commit contains a misuse, we determine the root cause of the problem and record its description as free text characteristics. We use these noted characteristics to later categorize our collected misuses using Wei et al. [57]'s taxonomy.

We also record the symptom that was reported to be observed. Finally, we refer to the library documentation to identify whether the API usage constraint that was violated is documented in the API. If the API directive is documented, we record the directive along with a link to the source.

3.2.3 Categorizing API misuses. We use closed coding [7] to label the confirmed misuses according to Wei et al. [57]'s taxonomy described in Section 2.2. Specifically, we use our recorded notes to determine each misuse's type (program element and violation), root cause, and symptom. We allow extensions to the taxonomy if we find any misuses that do not fit into one of the existing categories.

Table 3: Statistics of identified misuses.

Library	Data Source		Total
	SO	GH	
pandas	4	1	5 (10%)
NumPy	2	11	13 (27%)
scikit-learn	7	6	13 (27%)
Matplotlib	7	3	10 (20%)
seaborn	8	0	8 (16%)
Total	28	21	49 (100%)

4 CHARACTERISTICS OF API MISUSES IN DATA-CENTRIC LIBRARIES

We find that approximately 8% of the Stack Overflow posts and 5% of the GitHub commits we analyzed contain API misuses. In total, we identify 49 misuses, with 28 identified from Stack Overflow and 21 from GitHub. Our data set contains misuses of 45 unique APIs. Table 3 provides a summary of the number of misuses, and their distribution across the five libraries. We now describe the details of the misuses we found, using the taxonomy in Figure 2.

4.1 API misuse types

The red boxes in Figure 2 show the additions we made to the taxonomy. Specifically, we find misuses that occur due to violating the expected order of execution, which we add. We also observe that some misuses are identified by problems in the data being processed. For example, the misuse in Figure 1 is *data dependent*, because if the column 'color' passed to `hue` has categorical data, the API works correctly. Therefore, we extend the taxonomy to include data dependency as a dimension for describing misuse type.

We now discuss the types of API misuses we observe based on this updated taxonomy. Table 4 shows the distribution of misuse types in our data set compared to the deep learning misuse statistics [57], which are shaded gray. Since we do not consider deprecations as misuses in our study, the *outdated* column in Table 4 is empty for our data. The table also shows how many of the misuses are data dependent, for each misuse type. Overall, out of the 49 misuses, 27 (55%) are data dependent.

Wei et al. [57] found that 51% of the deep learning misuses involved API methods, 35% involved API parameters, while 13% involved API conditions. In our study, 43% of the misuses involve API methods, 51% involve API parameters, while 6% involve API conditions. The higher percentage of API parameter misuses in our study can be attributed to the diverse range of data types accepted by the libraries we analyzed, coupled with the specific constraints imposed on the data within a given context. For example, Matplotlib's `text` method accepts any data type for the `x` and `y` parameters, but the data type must match the previously set axes type. We now discuss the overall API misuse types we observe in terms of combination of program element and violation type.

Missing API Method: In general, missing an API call is one of the most commonly discussed type of API misuse in the literature [3, 11, 20, 48]. We observe that nine (18%) misuses of our studied libraries result from developers missing necessary calls to an API.

When examining misuses in deep learning applications, Wei et al. [57] found that failing to call APIs such as `flatten` causes shape mismatch errors. While shape mismatch errors are unique

Table 4: API misuse types of the data-centric libraries we study compared to deep learning libraries [57], shown in gray.

			Missing	Redundant	Replacement	Ordering	Outdated	**Total**
API method	**This study**	**Total**	9 (18%)	6 (12%)	4(8%)	2 (4%)	-	21 (43%)
		DD	4 (15%)	5 (19%)	1 (4%)	1 (4%)	-	11 (42%)
	DL only [57]		113 (13%)	138 (15%)	130 (15%)	-	74 (8%)	455 (51%)
API parameter	**This study**	**Total**	9 (18%)	1(2%)	15 (31%)	-	-	25 (51%)
		DD	5 (19%)	-	10 (37%)	-	-	15 (56%)
	DL only [57]		88 (10%)	56 (6%)	60 (7%)	-	115 (13%)	319 (36%)
API condition	**This study**	**Total**	3 (6%)	-	-	-	-	3 (6%)
		DD	1 (4%)	-	-	-	-	1 (4%)
	DL only [57]		43 (5%)	29 (3%)	17 (2%)	-	28 (3%)	117 (13%)

DD=data dependent, DL=deep learning

```
1 import seaborn as sns
2
3 tips = sns.load_dataset('tips')
4 - g = sns.FacetGrid(data=tips, col='time', row='sex')
5 - g.map(sns.lmplot, 'total_bill', 'tip')
6 + sns.lmplot(data=tips, x='total_bill', y='tip', col='time', row='sex')
```

Figure 3: Redundant API call to seaborn's `FacetGrid` when using `lmplot`

type of API misuse in deep learning libraries, as they are heavily reliant on tensor computations, we also observe similar misuses in our data set. For example, in pandas, failing to call `pivot` on a pandas dataframe before passing it to `heatmap` results in a runtime error, because the input is not in wide format as `heatmap` expects. It is important to note that this misuse is data dependent and only occurs when the data is in long format.

Redundant API Method: An API call is redundant if it is an API that the developer should not call in the given context and calling it would lead to an unexpected outcome or has no effect. Figure 3 shows an example of a redundant call to `FacetGrid`. Since `lmplot` internally uses `FacetGrid`, users could simply use `lmplot` (Line 6) without passing it to `FacetGrid` (Line 4, 5). In this case, the extra call is not harmless, using the combination shown in Figure 3 Line 4, 5 actually causes a run-time error. Notably, out of the six misuses in this category, five are data dependent.

API Method Replacement: We find four misuses (8%) where the developer used an incorrect API; only one of them are data dependent. For example, when using scikit-learn's `StandardScaler`, calling `transform` before it trains on a sample of data is incorrect. The correct method to call is `fit_transform` [50].

API Method Ordering: Some libraries expect APIs to be called in a specific order. For example, seaborn expects developers to set the axis tick labels *before* applying any formatting to the labels. Otherwise, the formatting is simply not applied. We find two misuses (4%) that belong to this category, with only one of them being data dependent. While Wei et al. [57] did not observe violation type related to call order, the FUM taxonomy [48] uses the label "Method call sequence" to signify misuses in this category.

Missing API Parameter: The second most common type of misuse is missing API parameter, representing 18% of our misuses. APIs can specify the parameters they accept, which can be required or optional, as well as keyword based or position based. When a developer does not pass a required parameter, Python would complain and issue a run-time error. Recall that we do not consider such cases since they are general Python programming errors that the interpreter can easily detect. However, there are cases where

```
1 import seaborn as sns
2
3 data = sns.load_dataset('tips')
4 - sns.distplot(data.tip, norm_hist=False)
5 + sns.distplot(data.tip, norm_hist=False, kde=False)
```

Figure 4: Missing API parameter for seaborn's `distplot`.

other code context implies the necessity of setting a particular parameter (or its value). This is what this API misuse type refers to.

When creating tensors in TensorFlow, the parameter `dtype` is optional. If a subsequent API requires a specific data type for the input tensor, failing to set `dtype` appropriately can lead to unexpected results and propagate errors throughout the program [57]. We observe a similar case in pandas where `replace(...)` [41] accepts either True or False for the parameter `inplace`. The default value of `inplace` is False, which means that `replace` would return a new data frame with the replaced values. Accordingly, a misuse occurs if the code does not have an assignment operation like `new_df = df.replace(...)` and the developer continues to use the old `df`, incorrectly assuming it has the replaced values. We also observe a case where the missing API parameter depends on another related parameter. For example, in Figure 4, the developer tries to set `norm_hist=False` (Line 4) without setting `kde=False`. Line 5 shows the correct setting of these related parameters. Without that, the y-axis show density instead of counts. We find that 19% of the data-dependent misuses are missing API parameter misuses, making it the second highest category in misuses that are data dependent.

API Parameter Replacement: We find that API parameter replacement is the most common type of misuse in our data (31%). This occurs when a method-call parameter is used incorrectly where a different value should be passed. Examples of incorrect usage includes passing incorrect data formats or unsupported values.

For instance, in Matplotlib's `datestr2num(...)` [26] method, the expected input date format is month-first (e.g., 04-21-2021). Providing a day-first format results in incorrect output due to improper conversion. Another example is when invoking `set_style(...)` [53] from seaborn to modify the aesthetics of a graph, developers can specify a set of key-value pairs (as a Python dictionary) from a predefined list of supported pairs. If an unsupported value is passed, seaborn silently ignores it, resulting in no change to the graph's appearance. As a last example, according to Matplotlib's documentation, calling `pyplot.text(x, y, s)` [28] to add a text label to a graph requires that the values of parameters `x` and `y` match the types of the corresponding axes (which would be determined from previous API calls while setting up the graph). A run-time error occurs when the argument type does not align with the axes type. Wei et al. [57] also observed API parameter replacement in deep learning APIs, where developers explicitly set the `dtype` parameter to `float32` to reduce computational costs.

We note that only 7% of the deep learning API misuses are API parameter replacements [57], versus 31% in our data. We believe that the reason for this difference is that deep learning libraries have a more restricted set of data types and values that can be passed to their APIs, while the libraries we studied (e.g., Matplotlib and seaborn) have a wider range of data types and values that can be passed to their APIs. We also note that 37% of API misuses that are data dependent belong to this misuse category.

Summary: API parameter replacement is the most common misuse type in our data, followed by missing API parameter. Misuses in these two categories account for 49% of the total misuses. Overall, we observe all API misuse types found in deep learning libraries, and also discover the new dimension of data-dependent misuses.

4.2 API Misuse Root Causes

We find that we could describe most of the root causes of our misuses using the existing taxonomy. However, we also observe the need for a new category of root causes, namely *configuration errors*, to allow us to capture 12 of our observed misuses. Table 5 shows the distribution of root causes of data-centric API misuses compared to deep learning API misuses.

Data conversion errors: Data conversion errors are the number one root cause of misuses of our data set (39%), while they were the second highest for deep learning misuses since they were preceded by device management errors. As an example, when utilizing visualization APIs such as seaborn's `lineplot`, providing column vectors for both the `x` and `y` parameters leads to a runtime error, as the API expects a 1D array. In another case, scikit-learn's `OneHotEncoder` requires the input dataframe column to be of uniform data type; otherwise, it cannot internally convert the data.

Null reference errors: Similar to findings of Wei et al. [57] (4%), we also observe a small percentage (10%) of null reference errors.

Algorithm errors: We observe a higher proportion (20%) of misuses whose root cause is algorithm errors. In the deep learning data set, the majority of misuses in this category were attributed to division by zero, often due to developers neglecting to pass a parameter (a small floating-point value) to mitigate such numerical errors [57]. In our data set, the algorithm problems varied. For example, scikit-learn's `roc_curve` API being incorrectly applied to multi-class classification, resulting in inaccurate results, even though the API is designed for binary classification.

Device management errors: In deep learning libraries, device management errors are the highest root cause, with the vast majority being specific to CPU vs. GPU usage. We observe only one device management error where a developer forgot to call `pyplot.close()` when plotting figures in a loop. We, expectedly, do not observe CPU/GPU usage related device management errors.

Configuration errors: There were many misuses whose root cause did not fit within the existing taxonomy. Accordingly, we derive this new category that refers to misuses resulting from incorrect or missing internally expected configuration settings within an API. For example, seaborn's `countplot` does not set the saturation values to the provided input values unless the `saturation` argument is set to 1; otherwise, an incorrect internal library/API configuration setting would be used. We find that approximately 24% of the misuses, mostly visualization API misuses, are due to such internally expected configurations.

Other: We find two misuses (4%) that did not clearly fall into any of the root cause categories. One is the misuse in Figure 3, where calling `FacetGrid` is redundant and problematic, because `lmplot` internally calls `FacetGrid`.

Table 5: Distribution of root causes of data-centric API misuses and deep learning API misuses [57].

Category	# (%) data-centric	# (%) deep learning [57]
Data conversion error	19 (39%)	246 (28%)
Algorithm error	10 (20%)	88 (10%)
Null reference error	5 (10%)	33 (4%)
Device management error	1 (2%)	337 (38%)
Configuration error	12 (24%)	-
Other errors	2 (4%)	10 (1%)

Summary: We find that data conversion errors in the additional data-centric libraries we study are even more prominent than deep learning libraries, while device management errors are naturally less. We also find a new category of root causes, *configuration errors*, encompassing 24% of the observed misuses.

4.3 API Misuse Symptoms

Program crashes: Similar to Wei et al. (36%) [57], we also find that program crashes is the most frequent misuse symptom (41%).

Unexpected output: Despite our smaller dataset, we observe a higher proportion of misuses that result in unexpected output (35%) compared to the 24% of API misuses found by Wei et al. [57]. Specifically, we find that 17 misuses did not result in any runtime errors but instead silently produced incorrect or unexpected outputs which could propagate through the program without a notice.

Out of the 17 misuses with unexpected output that we observe, 11 are related to APIs from the two visualization libraries while the remaining 6 instances are from scikit-learn and pandas. Figure 1b is an example of incorrect output resulting from misusing visualization library APIs. We also observe cases where misusing the visualization library's APIs resulted in incorrect axis tick labels, incorrect axis scales (count vs. normalized values), and incorrect plot styles such as not displaying an axis grid. The incorrect outputs caused by misusing APIs from non-visualization libraries varied. For example, when calling `replace()` on pandas dataframe, the API returns a dataframe with replaced values. Failing to assign the return dataframe produces incorrect output (that is harder to spot than an incorrect visualization for example), because the developers continue to process the dataframe with initial values.

Low efficiency: Only 20% of the misuses we observed result in low efficiency issues related to performance or memory usage. In contrast, Wei et al.'s findings [57] indicate that 32% of misuses in deep learning APIs lead to poor performance. We note that in their data, device management errors accounted for the majority of misuse root causes (38%), which explains why they find a higher proportion of misuses that result in low efficiency since configuring the correct device (i.e., CPU vs GPU) could result in faster training, considering the high computation cost of deep learning models. In our studied libraries, the low efficiency symptoms are related to the misuse of APIs that are computationally expensive, when specific configurations are not set. For example, when calling `intersect1d` in NumPy, the parameter `assume_unique` should bet set to `True` if the input arrays are unique, speeding up calculation.

Return warning: Only two misuses in our dataset resulted in warnings. One pandas misuse is due to chain indexing, while one Matplotlib misuse is caused by calling the `legend` function when there was no legend to display. We note that Wei et al. [57] observe much more warnings, because most of their observed warnings were deprecation warnings, which we do not consider.

Summary: Program crashes are the most prevalent symptom in data-centric libraries. However, we observe that our libraries exhibit a higher proportion of misuses that result in incorrect output.

4.4 Documented API Directives

We find that 39% of the misuses have documentation that explicitly states the correct usage of the API. Among the libraries that we analyzed, scikit-learn has the highest percentage of misuses with explicit API directives. We also analyze the relationship between misuse symptoms and the presence of explicit API directives. Only 20% of misuses resulting in low efficiency are accompanied by documented directives. In contrast, at least half of the misuses that result in unexpected output or warnings have explicitly documented API directives. These findings indicate that developers misuse data-centric APIs despite explicitly documented directives.

5 DISCUSSION

The goal of this empirical study is to understand whether the new types of API misuses, root causes, and symptoms found for deep learning libraries are specific to the deep learning domain or may extend to other libraries. Specifically, we investigated misuses of Python data processing, machine learning, and visualization APIs and categorized them using an existing deep learning misuse taxonomy [57]. We find that the misuses of these libraries exhibit similar behavior with those of deep learning libraries, with the heavy reliance on data being the common characteristic leading to many of the misuses. Furthermore, the multitude of parameters in data-centric APIs, which accept various data types and formats, contribute significantly to the occurrence of misuses. Our findings have implications for language and API design, surfacing information buried in documentation, and designing misuse detectors.

Implications for language support and API design: We find that 55% of the misuses in our data set are data dependent. Furthermore, 56% of these data-dependent misuses are due to violating various parameter/argument directives. For example, seaborn's `heatmap` method expects the input data in wide-format. Internally, the API checks if the input is a pandas dataframe or a rectangular array which then can be converted to a pandas dataframe. For this type checking, seaborn uses Python function `isinstance()`. Other than that, the API has no further verification to check if the input is in the correct format. Python does not provide any tools that can help library designers enforce such restrictions. Newer Python versions (after 3.5) did introduce type hints with the *typing* module, which can then be used for static type checking by external tools such as *mypy* [33]). However, type checking alone cannot infer the *content* of various data structures. While third-party library APIs typically include checks for data format (e.g., pandas dataframe or NumPy array), ensuring the integrity of data content, such as detecting mixed data types within a column of a pandas dataframe, poses a challenge in API design. Programming languages like Java have annotation frameworks (e.g., Java Checker Framework or IntelliJ IDEA's Java Annotations) that enforce API contracts such as parameters not being null, being within a certain range, or in a specific format like an email; however, there is limited support for inter-parameter dependencies, especially if it is dependent on data. Overall, our findings imply that programming language designers need to provide tools that can be used to enforce necessary restrictions when designing data-centric third-party APIs.

Implications for API documentation: Approximately half of the misused APIs did not have documented API directives, which could potentially have led to their misuse. For example, Matplotlib's `datestr2num(...)` expects the input string to be in the date-first format (e.g., 04-21-2021) even though the documentation does not explicitly state this. The documentation, however, notes that Matplotlib uses the *dateutil* library to parse a string to date. Unless developers investigate the documentation of *dateutil*, it is not possible to know the assumptions made by API designers of `datestr2num(...)`.

While missing documentation or hard-to-navigate documentation is not specific to data-centric libraries per se [25, 30, 47], we speculate that the specialized domain knowledge of using these libraries might make it more problematic as evidenced by the high misuse rate of documented API directives in our data set. Thus, we encourage data-centric library authors to explicitly document all API directives, especially those related to expected data formats and not to assume any domain knowledge on behalf of the users. Researchers may also help by enhancing previously proposed techniques for identifying and surfacing critical information in documentation [47], or even augmenting documentation to include missing information [55]. Our data set of API misuses along with any documented API directives can provide a starting point for further pursuing this research direction.

Implications for misuse detectors: Wei et al. [57] showed that existing misuse detectors are not effective in detecting deep learning API misuses. Our study shows that the characteristics of deep learning API misuses extend to data-centric APIs, and that parameter misuse and furthermore data-dependent misuses are even more prevalent in these libraries. To detect data-centric API misuses, tool developers need to not only consider dynamic analysis but to develop techniques that relate the format/content of data to the API calls and parameter values in the code. We note that tools such as *Data Linter* [14] provides summary statistics of data and trigger notifications for incorrectly inferred types in data (e.g., string instead of numeric). While Data Linter allows developers to preprocess a data set before they feed to a program, any modifications or data manipulation that happens during the program execution is out of its scope. Our detailed analysis of the misuses we found and the data set we provide can help tool developers build more advanced misuse detectors.

6 THREATS TO VALIDITY

Internal validity: Internal validity threats relate to the degree to which the study's outcomes are attributed to the methodology, minimizing personal bias. We followed a manual labeling approach, and any manual analysis is prone to subjectivity. To reduce subjectivity, we iteratively developed a stable coding guideline by having three annotators annotate each thread in ∼43% of our Stack Overflow thread sample. Once we stabilized the coding guideline, we continued with one annotator per data point for the GitHub commits. To further reduce subjectivity, we created minimal reproducible examples for each misuse and conservatively eliminated any misuses that we could not reproduce.

Construct validity: Construct validity threats relate to the extent to which our study reflects the concept of API misuse within the defined scope. Approximately 8% of the both Stack Overflow threads and the GitHub commits that we analyzed contained API misuses. While this ratio matches previous misuse studies [1, 11], we note that Wei et al. [57] found misuses in a larger portion of their analyzed commits (21%). One notable differences is that the libraries that we study do not rely on hardware devices, a characteristic common in deep learning libraries. Additionally, we do not consider deprecation handling as API misuse. These two categories account for approximately 58% of the misuses found in Wei et al.'s data [57]. We note that we did find 48 commits that fix deprecated API usage that we did not account as misuses, which would have largely increased our data set size. Overall, our smaller total number of misuses is likely due to our stricter API misuse definition discussed in Section 2, but which is more aligned with our goal of observing misuses within the domain of data-centric libraries.

External validity: External validity threats relate to the generalizability of our results. We selected 5 popular Python libraries to study the misuse types in data-centric applications. We did not study all data-centric libraries in Python or those in other programming languages. Thus, our results may not generalize beyond the selected libraries. However, our selection represents more domain diversity than the typical deep learning focus [15, 57, 60] and extends even beyond machine learning in general. We encourage researchers to further study more data-centric libraries. We also prioritized finding commits for diverse APIs, which may mean that we missed observing more instances of the same or similar misuses.

7 RELATED WORK

Types of API Misuse: Amann et al. [1] created a data set of Java API misuses known as MuBench. In followup work, the authors used this data to create a taxonomy for classifying Java API misuses (MuC) [3]. By analyzing API documentation of JDK, JFace, and Java Commons Collection, Monperrus et al. [30] developed a taxonomy of types of API directives. Schlichtig et al. [48]'s API directive definition that we use in Section 2 is inspired from Monperrus et al.'s work. Schlichtig et al. [48]'s framework for classifying API misuses is probably the most comprehensive API misuse classification taxonomy/framework, mainly derived from the Java literature.

He et al. [11] created a taxonomy (PUM) for Python misuses, comparing it to MuC and FUM above. Their work provides valuable insights of the impact of the dynamic features of Python on using APIs. The deep learning API misuse taxonomy by Wei et al. [57] is based on these previous API misuse taxonomies while extended to account for unique characteristics of the deep learning domain [3, 11, 20, 48]. In our study, we utilize the deep learning taxonomy to categorize misuses of data-centric API in general, with certain modifications made to the categories to enhance expressiveness.

API directives: A library's documentation contains information on how to use an API. While some of the information is obvious and thus has a little value [25], some information is critical for the correct program behavior such as call order and condition checking [45]. Due to the large amount of textual details, developers could fail to notice important information when learning a new API [25]. To alleviate this problem, Robillard and Chhetri proposed a technique to automatically identify important knowledge items in documentation by analyzing linguistic patterns in sentences [47]. Monperrus et al. [30]'s taxonomy of Java API directives is based on analyzing documentation. We do not systematically analyze documentation, but we look for documented API directives specifically related to the misuses we observed. Our results showed that the majority of the misused APIs (39%) have documented directives, emphasizing the need for additional support for highlighting and surfacing this information to developers.

API misuse detection: There is a long line of research that aims to detect API misuses. Early API misuse detectors mined frequent usage patterns from source code to identify correct usages; deviations from those mined patterns are then considered violations [2, 21, 24, 31, 36, 43, 56]. Pattern-based techniques suffer from low precision and low recall, because these tools are limited to detecting only the frequently occurring usages. To remedy this, researchers developed detectors to include correct usages automatically captured from documentation and client projects [22, 45, 59]. Additionally, some detectors use domain specific language (DSL) to detect API misuses where usage constraints are manually specified by an expert or automatically extracted from documentation [9, 10, 17, 19]. There is also recent work that demonstrates the possibility of using large language models (LLMs) for misuse detection [57]. In this paper, our goal is not to design a misuse detector but to understand the nature of API misuses of data-centric libraries, paving the way for future research to use our findings to design and develop detection tools. For example, we find that 55% of misuses in our data set are data dependent, suggesting that tool builders need to account for the data driven nature of certain libraries when designing API misuse detectors for these libraries.

Reasons for misusing APIs: There is also a line of work that aims to understand the difficulties developers face when using APIs [12, 29, 34, 46] as well as general API usability problems [32]. While we do not investigate *why* developers misuse APIs of data-centric libraries, our results (e.g., undocumented API directives) suggest potential API usability issues.

8 CONCLUSION

The idea for this empirical study started with our conjecture that many of the recently observed API misuses of deep learning libraries are not necessarily specific to machine learning or deep learning per se but are actually due to the data-centric nature of these libraries. accordingly, we study API misuse in five additional data-centric libraries and indeed find similar API misuse characteristics. We find that more than half of the misuses (55%) are data-dependent and that API parameter misuse is even more prevalent than deep learning libraries. While run-time errors are the most observed symptom, incorrect output comes as a close second. Data-conversion errors emerge as the primary root cause of the misuses we observed. Surprisingly, even when APIs were accompanied by explicit API directives, developers still encountered difficulties, suggesting challenges in extracting relevant information from textual documentation. Reflecting on our findings, we discuss implications for language and API design, misuse detectors, and surfacing buried information in documentation. Our in-depth analysis of data-centric API misuses, along with the data we share containing the misuses and their (documented) API directives, paves the way for building better support tools and misuse detectors.

REFERENCES

[1] Sven Amann, Sarah Nadi, Hoan Anh Nguyen, Tien N. Nguyen, and Mira Mezini. 2016. MUBench: a benchmark for API-misuse detectors. In *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*. ACM, 464–467. https://doi.org/10.1145/2901739.2903506

[2] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N. Nguyen, and Mira Mezini. 2019. Investigating next steps in static API-misuse detection. In *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*. IEEE / ACM, 265–275. https://doi.org/10.1109/MSR.2019.00053

[3] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N. Nguyen, and Mira Mezini. 2019. A Systematic Evaluation of Static API-Misuse Detectors. *IEEE Trans. Software Eng.* 45, 12 (2019), 1170–1188. https://doi.org/10.1109/TSE.2018.2827384

[4] Anonymous. 2024. *ESEM 2024 - Data-centric API misuse data set.* https://figshare.com/s/0062f28e8587fe9ce715

[5] Wilson Baker, Michael O'Connor, Seyed Reza Shahamiri, and Valerio Terragni. 2022. Detect, Fix, and Verify TensorFlow API Misuses. In *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, HI, USA, March 15-18, 2022*. IEEE, 925–929. https://doi.org/10.1109/SANER53432.2022.00110

[6] Houssem Ben Braiek, Foutse Khomh, and Bram Adams. 2018. The open-closed principle of modern machine learning frameworks. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, Andy Zaidman, Yasutaka Kamei, and Emily Hill (Eds.). ACM, 353–363. https://doi.org/10.1145/3196398.3196445

[7] Kathy Charmaz. 2014. Constructing grounded theory. (2014).

[8] Rosa Falotico and Piero Quatto. 2015. Fleiss' kappa statistic without paradoxes. *Quality & Quantity* 49, 2 (2015), 463–470.

[9] Zuxing Gu, Jiecheng Wu, Chi Li, Min Zhou, Yu Jiang, Ming Gu, and Jiaguang Sun. 2019. Vetting API usages in C programs with IMChecker. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. IEEE / ACM, 91–94. https://doi.org/10.1109/ICSE-Companion.2019.00046

[10] Mansur Gulami. 2022. *A Human-in-the-loop Approach to Generate Annotation Usage Rules.* Master's thesis.

[11] Xincheng He, Xiaojin Liu, Lei Xu, and Baowen Xu. 2023. How Dynamic Features Affect API Usages? An Empirical Study of API Misuses in Python Programs. In *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2023, Taipa, Macao, March 21-24, 2023*, Tao Zhang, Xin Xia, and Nicole Novielli (Eds.). IEEE, 522–533. https://doi.org/10.1109/SANER56733.2023.00055

[12] Daqing Hou and Lin Li. 2011. Obstacles in Using Frameworks and APIs: An Exploratory Study of Programmers' Newsgroup Discussions. In *The 19th IEEE International Conference on Program Comprehension, ICPC 2011, Kingston, ON, Canada, June 22-24, 2011*. IEEE Computer Society, 91–100. https://doi.org/10.1109/ICPC.2011.21

[13] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of real faults in deep learning systems. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*. ACM, 1110–1121. https://doi.org/10.1145/3377811.3380395

[14] Nick Hynes, D Sculley, and Michael Terry. 2017. The data linter: Lightweight, automated sanity checking for ml data sets. In *NIPS MLSys Workshop*, Vol. 1.

[15] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A comprehensive study on deep learning bug characteristics. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. ACM, 510–520. https://doi.org/10.1145/3338906.3338955

[16] kaggle. 2023. State of Data Science and Machine Learning 2022. https://www.kaggle.com/kaggle-survey-2022

[17] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, and Ram Kamath. 2017. CogniCrypt: supporting developers in using cryptography. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. IEEE Computer Society, 931–936. https://doi.org/10.1109/ASE.2017.8115707

[18] J. Richard Landis and Gary G. Koch. 1977. The Measurement of Observer Agreement for Categorical Data. *Biometrics* 33, 1 (1977), 159–174. http://www.jstor.org/stable/2529310

[19] Chi Li, Zuxing Gu, Min Zhou, Jiecheng Wu, Jiarui Zhang, and Ming Gu. 2019. API Misuse Detection in C Programs: Practice on SSL APIs. *Int. J. Softw. Eng. Knowl. Eng.* 29, 11&12 (2019), 1761–1779. https://doi.org/10.1142/S0218194019400205

[20] Xia Li, Jiajun Jiang, Samuel Benton, Yingfei Xiong, and Lingming Zhang. 2021. A Large-scale Study on API Misuses in the Wild. In *14th IEEE Conference on Software Testing, Verification and Validation, ICST 2021, Porto de Galinhas, Brazil, April 12-16, 2021*. IEEE, 241–252. https://doi.org/10.1109/ICST49551.2021.00034

[21] Zhenmin Li and Yuanyuan Zhou. 2005. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*. ACM, 306–315. https://doi.org/10.1145/1081706.1081755

[22] Qingmi Liang, Zhirui Kuai, Yangqi Zhang, Zhiyang Zhang, Li Kuang, and Lingyan Zhang. 2022. MisuseHint: A Service for API Misuse Detection Based on Building from Documentation and Codebase. In *IEEE International Conference on Web Services, ICWS 2022, Barcelona, Spain, July 10-16, 2022*. IEEE, 246–255. https://doi.org/10.1109/ICWS55610.2022.00046

[23] Yunkai Liang, Yun Lin, Xuezhi Song, Jun Sun, Zhiyong Feng, and Jin Song Dong. 2022. gDefects4DL: A Dataset of General Real-World Deep Learning Program Defects. In *44th IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2022, Pittsburgh, PA, USA, May 22-24, 2022*. ACM/IEEE, 90–94. https://doi.org/10.1145/3510454.3516826

[24] Christian Lindig. 2015. Mining Patterns and Violations Using Concept Analysis. In *The Art and Science of Analyzing Software Data*, Christian Bird, Tim Menzies, and Thomas Zimmermann (Eds.). Morgan Kaufmann / Elsevier, 17–38. https://doi.org/10.1016/b978-0-12-411519-4.00002-1

[25] Walid Maalej and Martin P. Robillard. 2013. Patterns of Knowledge in API Reference Documentation. *IEEE Trans. Software Eng.* 39, 9 (2013), 1264–1282. https://doi.org/10.1109/TSE.2013.12

[26] Matplotlib. 2023. datestr2num. https://matplotlib.org/stable/api/dates_api.html#matplotlib.dates.datestr2num

[27] Matplotlib. 2023. Matplotlib. https://matplotlib.org/

[28] Matplotlib. 2023. pyplot.text. https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.text.html

[29] Michael Meng, Stephanie Steinhardt, and Andreas Schubert. 2018. Application Programming Interface Documentation: What Do Software Developers Want? *Journal of Technical Writing and Communication* 48, 3 (2018), 295–330. https://doi.org/10.1177/0047281617721853

[30] Martin Monperrus, Michael Eichberg, Elif Tekes, and Mira Mezini. 2012. What should developers be aware of? An empirical study on the directives of API documentation. *Empir. Softw. Eng.* 17, 6 (2012), 703–737. https://doi.org/10.1007/s10664-011-9186-4

[31] Martin Monperrus and Mira Mezini. 2013. Detecting missing method calls as violations of the majority rule. *ACM Trans. Softw. Eng. Methodol.* 22, 1 (2013), 7:1–7:25. https://doi.org/10.1145/2430536.2430541

[32] Brad A. Myers and Jeffrey Stylos. 2016. Improving API usability. *Commun. ACM* 59, 6 (2016), 62–69. https://doi.org/10.1145/2896587

[33] mypy. 2023. mypy. https://mypy-lang.org/

[34] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. 2016. Jumping through Hoops: Why Do Java Developers Struggle with Cryptography APIs?. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 935–946. https://doi.org/10.1145/2884781.2884790

[35] Sarah Nadi and Christoph Treude. 2020. Essential Sentences for Navigating Stack Overflow Answers. In *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020*, Kostas Kontogiannis, Foutse Khomh, Alexander Chatzigeorgiou, Marios-Eleftherios Fokaefs, and Minghui Zhou (Eds.). IEEE, 229–239. https://doi.org/10.1109/SANER48275.2020.9054828

[36] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. 2009. Graph-based mining of multiple object usage patterns. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*. ACM, 383–392. https://doi.org/10.1145/1595696.1595767

[37] NumPy. 2023. NumPy. https://numpy.org/

[38] Query Stack Overflow. 2023. Stack Exchange Data Explorer. https://data.stackexchange.com/stackoverflow/query/

[39] Stack Overflow. 2024. Survey. https://survey.stackoverflow.co/2023/#overview

[40] pandas. 2023. pandas. https://pandas.pydata.org/

[41] pandas. 2023. repalce. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.replace.html

[42] PyDriller. 2023. PyDriller documentation. https://pydriller.readthedocs.io/en/latest/index.html

[43] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. 2007. Static specification inference using predicate mining. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. ACM, 123–134. https://doi.org/10.1145/1250734.1250749

[44] Sebastian Raschka, Joshua Patterson, and Corey Nolet. 2020. Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence. *Information* 11, 4 (2020), 193. https://doi.org/10.3390/INFO11040193

[45] Xiaoxue Ren, Xinyuan Ye, Zhenchang Xing, Xin Xia, Xiwei Xu, Liming Zhu, and Jianling Sun. 2020. API-Misuse Detection Driven by Fine-Grained API-Constraint Knowledge Graph. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 461–472. https://doi.org/10.1145/3324884.3416551

[46] Martin P. Robillard. 2009. What Makes APIs Hard to Learn? Answers from Developers. *IEEE Softw.* 26, 6 (2009), 27–34. https://doi.org/10.1109/MS.2009.193

[47] Martin P. Robillard and Yam B. Chhetri. 2015. Recommending reference API documentation. *Empir. Softw. Eng.* 20, 6 (2015), 1558–1586. https://doi.org/10.1007/s10664-014-9323-y

[48] Michael Schlichtig, Steffen Sassalla, Krishna Narasimhan, and Eric Bodden. 2022. FUM - A Framework for API Usage constraint and Misuse Classification. In *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, HI, USA, March 15-18, 2022*. IEEE, 673–684. https://doi.org/10.1109/SANER53432.2022.00085

[49] scikit learn. 2023. scikit-learn. https://scikit-learn.org/stable/

[50] scikit learn. 2023. StandardScaler. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html#sklearn.preprocessing.StandardScaler.fit_transform

[51] seaborn. 2023. scatterplot. https://seaborn.pydata.org/generated/seaborn.scatterplot.html

[52] seaborn. 2023. seaborn. https://seaborn.pydata.org/

[53] seaborn. 2023. set_style. https://seaborn.pydata.org/generated/seaborn.set_style.html

[54] Suresh Thummalapenta and Tao Xie. 2009. Alattin: Mining Alternative Patterns for Detecting Neglected Conditions. In *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*. IEEE Computer Society, 283–294. https://doi.org/10.1109/ASE.2009.72

[55] Christoph Treude and Martin P. Robillard. 2016. Augmenting API documentation with insights from stack overflow. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, Laura K. Dillon, Willem Visser, and Laurie A. Williams (Eds.). ACM, 392–403. https://doi.org/10.1145/2884781.2884800

[56] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. 2007. Detecting object usage anomalies. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*. ACM, 35–44. https://doi.org/10.1145/1287624.1287632

[57] Moshi Wei, Nima Shiri Harzevili, YueKai Huang, Jinqiu Yang, Junjie Wang, and Song Wang. 2024. Demystifying and Detecting Misuses of Deep Learning APIs. In *Proceedings of the 46th International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14 - 20, 2024*. IEEE, 1–13.

[58] Ming Wen, Yepang Liu, Rongxin Wu, Xuan Xie, Shing-Chi Cheung, and Zhendong Su. 2019. Exposing library API misuses via mutation analysis. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. IEEE / ACM, 866–877. https://doi.org/10.1109/ICSE.2019.00093

[59] Hushuang Zeng, Jingxin Chen, Beijun Shen, and Hao Zhong. 2021. Mining API Constraints from Library and Client to Detect API Misuses. In *28th Asia-Pacific Software Engineering Conference, APSEC 2021, Taipei, Taiwan, December 6-9, 2021*. IEEE, 161–170. https://doi.org/10.1109/APSEC53868.2021.00024

[60] 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, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, Frank Tip and Eric Bodden (Eds.). ACM, 129–140. https://doi.org/10.1145/3213846.3213866