

## What Kinds of Contracts Do ML APIs Need?

Samantha Syeda Khairunnesa · Shibbir  
Ahmed · Sayem Mohammad Imtiaz ·  
Hridesh Rajan · Gary T. Leavens

Received: date / Accepted: date

**Abstract** Recent work has shown that Machine Learning (ML) programs are error-prone and called for contracts for ML code. Contracts, as in the design by contract methodology, help document APIs and aid API users in writing correct code.

The question is: what kinds of contracts would provide the most help to API users? We are especially interested in what kinds of contracts help API users catch errors at earlier stages in the ML pipeline. We describe an empirical study of posts on *Stack Overflow* of the four most often-discussed ML libraries: *TensorFlow*, *Scikit-learn*, *Keras*, and *PyTorch*. For these libraries, our study extracted 413 informal (English) API specifications. We used these specifications to understand the following questions. What are the root causes and effects behind ML contract violations? Are there common patterns of ML contract violations?

When does understanding ML contracts require an advanced level of ML software expertise? Could checking contracts at the API level help detect the violations in early ML pipeline stages? **Our key findings are that the most**

---

Samantha Syeda Khairunnesa  
Dept. of Computer Science and Information Systems, Bradley University  
E-mail: skhairunnesa@fsmail.bradley.edu

Shibbir Ahmed  
Dept. of Computer Science, Iowa State University  
E-mail: shibbir@iastate.edu

Sayem Mohammad Imtiaz  
Dept. of Computer Science, Iowa State University  
E-mail: sayem@iastate.edu

Hridesh Rajan  
Dept. of Computer Science, Iowa State University  
E-mail: hridesh@iastate.edu

Gary T. Leavens  
Dept. of Computer Science, University of Central Florida  
E-mail: Leavens@ucf.edu

commonly needed contracts for ML APIs are either checking constraints on single arguments of an API or on the order of API calls. The software engineering community could employ existing contract mining approaches to mine these contracts to promote an increased understanding of ML APIs. We also noted a need to combine behavioral and temporal contract mining approaches. We report on categories of required ML contracts, which may help designers of contract languages.

**Keywords** Machine Learning · API contracts · Empirical software engineering · Software engineering for machine learning

## 1 Introduction

Software developers are increasingly integrating machine learning (ML) into systems using ML libraries’ application programming interfaces (APIs). However, ML software is bug-prone Zhang et al. (2018b); Islam et al. (2019); Humbatova et al. (2020) and like traditional software could benefit from adopting a design-by-contract methodology Islam et al. (2019). Contracts can specify the expected behavior of an API and help client code use the API correctly, e.g., a contract might require that the `fit` method be applied to a model before calling the `predict` method. Another example can be given using the `MaxPooling2D` method for retaining the most prominent features of the feature map in a convolutional neural network (CNN). There is a contract on the `MaxPooling2D` method’s argument, `data_format`, based on the shape of the input image. If the input image has the shape (N, C, H, W), then the value for the argument `data_format` is set to `channels_first`. If the input has the shape (N, H, W, C), then `data_format` must be set to `channels_last`. Here, the letters N, H, W, and C represent the following: the number of images in the batch, the height of the image, the width of the image, and the number of channels of the image.

There is a rich body of prior work that can be grouped into two categories: work on contracts for non-ML software and work on ML software.

The first category, contracts for non-ML software, can be further divided into two types: behavioral and temporal. Behavioral contracts Hoare (1969); Meyer (1988); Pradel and Gross (2009); Păsăreanu and Rungta (2010); Nguyen et al. (2014); Khairunnesa et al. (2017) specify acceptable program states, typically for calls to individual methods in an API. For instance, in the Java Development Kit (JDK) `String` class, the precondition `beginIndex ≤ endIndex` must be true before calling method `substring(beginIndex, endIndex)`. The contracts that belong to this category are preconditions (as in the example), or postconditions (constraints ensured by the execution of the call) for a method in question. There are also class invariants that capture the constraints for all methods in a particular class. Temporal contracts Manna and Pnueli (1992); Gruska et al. (2010); Nguyen et al. (2009); Wasylkowski et al. (2007) encode the correct ordering of calls, possibly among multiple APIs. For example, in

Python, after creating a `threading.Lock` object, once a thread makes a call to `Lock.acquire()`, that thread should eventually call `Lock.release()`.

The notion of contracts in this study is similar to the kinds of contracts described just before this phrase. We have used the same definition of (behavioral and temporal) contracts in this study. A contract specifies the correct usage of an API and an incorrect usage is a contract violation.

The second category is about ML software and its bugs Zhang et al. (2018b); Islam et al. (2019); Humbatova et al. (2020) and bug fixes Sun et al. (2017); Islam et al. (2020). These works study either the implementation of ML library APIs or usage information about those APIs. Zhang et al. (2018b) and Humbatova et al. (2020) focused on understanding the defects in different ML libraries. The authors (Zhang et al. (2018b)) noted that the defect might come from various sources, e.g., program code, execution environment, library framework itself, etc. In contrast, the focus of this study is to gain an understanding of ML API contracts. Islam et al. (2019) reported on API misuse. API misuse can be detected if contract obligations are specified. Sun et al. (2017) investigated the issues in various ML libraries to understand the bug-fix patterns in these libraries, whereas Islam et al. (2020) studied the deep neural network (DNN) models to understand the bug-fix patterns. In our study, we focused on ML API contracts and corresponding breaches. Suppose a user maintains a contract obligation for an ML API. In that case, if the API demonstrates exceptional behavior upon exiting, the issue may be present in the implementation of the API.

Our work focuses on investigating the kinds of contracts required to establish the correct usage of ML APIs. The main question is: *what are the kinds of contracts required to establish the correct usage of ML APIs?* We observe that ML software is different from traditional software in several ways. In ML software, problem-solving is largely dependent on training data and subject to precise settings of hyper-parameters Zhang et al. (2018b). A prior work by Humbatova et al. (2020) suggested that choice of loss function/optimizer, missing/redundant/wrong layers, etc. are distinctive bugs in ML software. Also, incorrect use of ML APIs may not always lead to crashes, but may instead lead to slower performance or statistically invalid results. In this study, we did not aim to check the reliability of the ML systems. Instead, we looked at the errors occurring in ML programs due to the incorrect usage of ML APIs.

We studied four popular ML libraries: *TensorFlow*, *Scikit-learn*, *Keras* and *PyTorch* and studied posts from the Q&A forum *Stack Overflow (SO)* that contain one of these libraries in a tag. The dataset (labeled *SO* posts, queries, source codes, etc.) generated during our study are available in the *figshare* repository, <https://figshare.com/s/c288c02598a417a434df>. This dataset includes a total of 1565 posts, from which we manually curated posts that hold 413 contracts for relevant ML APIs. We use this data to answer the following research questions:

**RQ1 (Root Cause and Effect):** What are the root causes and effects behind ML contract violations?

**RQ2 (Patterns):** Are there common patterns of ML contract violations?

**RQ3 (Contract Comprehension Challenges):** When does understanding ML contracts require an advanced level of ML software expertise?

**RQ4 (Contract Violation Detection):** Can checking contracts at the API level help detect the violation in early ML pipeline stages?

These questions, and the data that support their answers, help to answer the main question, i.e., they enable researchers and practitioners to pinpoint where immediate support is required in terms of contracts for ML APIs. The key findings from our study are summarized in Table 1.

**Table 1:** Findings and Insights

RQ	Findings	Actionable Insight
RQ1	Most frequent contracts for ML APIs: (§3.1.1 ) 1. Constraint check on single arguments of an API. 2. Order of API calls that become a requirement eventually.	This is a good news because the software engineering (SE) community can employ some existing contract mining approaches to also mine contracts for ML APIs; but there might be a need to combine behavioral and temporal contract mining approaches that have been independently developed thus far.
RQ4	ML API contracts that are commonly violated occur in earlier ML pipeline stages (§3.4).	A verification system with ML contract knowledge can explain whether a bug in the ML system that used those APIs stemmed from an API contract breach.
RQ3	The absence of precise error messages (§3.1.2) due to system failures makes contract comprehension and violation detection more challenging.	As domain experts can understand the challenging ML contracts (§3.3), this knowledge encoded as contracts can enable improved debugging mechanisms.
RQ1	ML APIs require several type checking contracts specific to ML (§3.1.1) and interdependency (Table 6) between behavioral and temporal contracts.	Programming methodology and tools for design by contract should include sufficient expressiveness for these additional types of contracts seen in ML APIs.

The contributions of our paper are the following. We provide a taxonomy for ML API contracts and corresponding root causes. This taxonomy (§2.3) added five new leaf node categories of contracts (with respect to the leaf categories observed in traditional behavioral and temporal contracts) observed in our study. The work also identified the stages of ML pipelines in which the violations occur (API contract violation locations) or affect the software and presented a dedicated classification (§2.4). To our knowledge, this is the first work that attempts to understand the types of required contracts needed to prevent problems that may arise when using these ML APIs in software systems. In §3, in addition to answering the research questions, we analyze the outcomes related to contract breaches. Finally, we provide recommendations to researchers, consumers, and producers of ML APIs based on the findings.

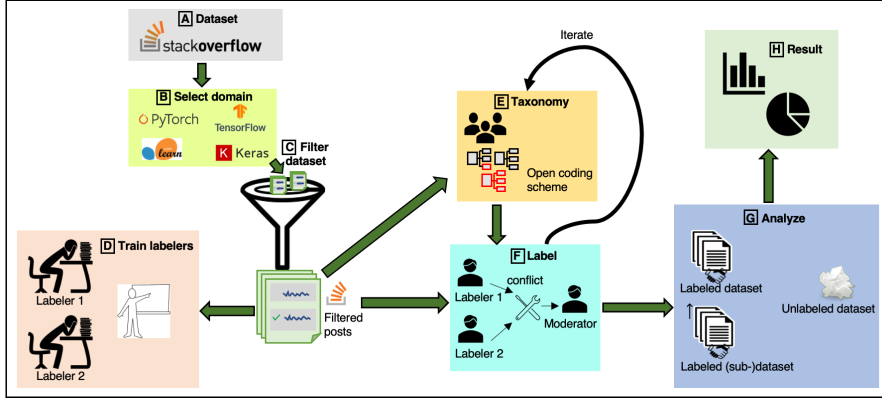


Fig. 1: Overview of the adopted methodology

## 2 Methodology

### 2.1 Overview

[A] In this study, we used Stack Overflow (*SO*) to investigate API contracts’ requirements for the most-asked about and widely-used ML libraries and frameworks. *SO* is a forum for software development professionals and enthusiasts. In recent years *SO* has served as an open repository for conducting studies on software engineering topics Zhang et al. (2018a); Cai et al. (2019); Aghajani et al. (2019); Beyer and Pinzger (2014); Barua et al. (2012); Rosen and Shihab (2015); Cummaudo et al. (2020). *SO*, as a forum, maintains a strict moderation policy, promotes a peer-reviewing mechanism, and incorporates a reward system for encouraging quality answers from the software developers Stack-Overflow Reputation (2023). Moreover, it has a vibrant user community and includes software developers from all walks of life, experiences, etc. StackOverflow Survey (2017). As a result, it offers a wealth of well-vetted information on numerous software development topics. As such, *SO* makes an excellent source for our study, as the primary goal of this study is to derive ML contracts from peer-reviewed and well-vetted content for the reliability of the findings. To capture the contracts, analyzing the large code corpus of API usages Wasylkowski et al. (2007); Nguyen et al. (2009); Khairunnesa et al. (2017) or the implementation of the software itself Cousot et al. (2013) are both well-known techniques. Our chosen methodology is closer to the former.

[B] We used the *SO* forum’s tags to identify the relevancy of a post to an ML library; if the question’s tag contained an ML library name, it was considered a post related to that library and was thus a candidate to be studied in this work.

We ranked the top ML libraries using the *frequency of these tags*, resulting in these four as the object of our study: *TensorFlow*, *Scikit-learn*, *Keras*, and *PyTorch*.

[C] Next we filtered these posts based on a set of defined criteria that are described in detail in §2.2.

[D] The second and third authors (labelers), both with a strong background in ML, were given background information on contract literature. Then they were given hands-on training with sample SO posts as described in §2.6.

[E] After the training process, 10% of the filtered dataset is used by the first three authors to develop the taxonomies used to label the filtered posts. Two iterations were needed to propose the final taxonomy presented here. The process is described in detail in §2.5.

[F] Next, these labelers identified contracts implicitly present in *SO* posts. We obtained 162, 122, 103, and 26 contracts, respectively, from the previously curated posts. Table 2 shows a summary of the dataset for each library in our study. For each *SO* question, we used the taxonomy of contracts (including proposed categories) from §2.3 to investigate the available information from the question and accepted answer to decide the type of contract obligation missing in the question and marked in the response. Hence, if the *SO* response describes the correct way of using an API of interest violated in the question, we identify that as a contract for the API that was implicitly present in *SO* posts. We have also used the taxonomies presented in §2.4 and §2.5 to complete the labeling in this stage. In §2.3, we describe the process of identifying the contract violation and potential contract for ML APIs with example *SO* posts from our study. Then, the first author, with expertise (of approx. 6 years) in contracts, reviewed the identified contracts and the *SO* post the contracts were extracted from. This served two purposes: it ensured that the identified contracts were correct and helped to reduce the threat of missing contracts that was implicitly present in the dataset from the second and third authors. The first author found only a handful of contracts missed by these two labelers. However, these missing contracts were found by at least one of the two labelers. Therefore, we did not note any new contracts this way. If one labeler identified a contract and the other did not, as they performed their labeling using the proposed taxonomy, this was identified as one of the reasons behind creating a conflict between the two labelers. We discussed in §2.6 how we resolved the conflicts in our study.

[G] As the labeling process is completed, we analyze our labeled dataset. Additionally, we have created a separately filtered dataset (a subset of the original) based on the question scores and analyzed questions with a relatively high score (in the range of 30-339). The intuition behind further separating these posts is that an author may ask one question, and only a handful of ML API users might run into it. Then another *SO* question may be inquired by someone but up-voted by hundreds of others who have the same problem. Thus, the intuition behind further separating these filtered posts was to understand how many ML API users are struggling with each problem. This separate dataset was compared against the entire dataset to be vigilant about the representative issues and respective conclusions we draw from the posts in §3. This subset is selected with the following criteria: select high-quality posts and keep manual efforts manageable. To that end, to ensure high-quality posts, we

select posts having better than average scores (avg. 18.9). To keep the manual effort manageable, we find a trade-off between sample size and its statistical power. We specifically choose 30 as a cut-off to have reasonable confidence in this additional study while keeping manual efforts manageable (about 90% confidence level with a 5% margin of error), resulting in 222 posts. We discarded posts if they did not capture any information regarding correct usage of ML APIs. Additionally, we grouped the discarded *SO* posts that we could not label as containing contracts during the manual curation step. We were unable to label some posts due to the following reasons: posts asking general clarification questions, unresolved issues with specific APIs of interest, an API unidentified in a post, a solution involving tuning, or a dependency between an unrelated API and a related API. For instance, in some of these posts, the ML API users is usually curious about performing a task at hand or inter-library code transformation and refactoring. To illustrate, in one *SO* post<sup>1</sup>, the author is knowledgeable that they can use the `summary()` API from Keras to print a model summary. Yet, they want to know how to do the same using Pytorch. Even if these posts are of interest to ML API users, these do not fall into the category of contract requirements. In our study, the number of total unlabelled posts is 1159, and the number of total unlabelled posts with a score of or higher 30 is 161.

[H] Finally we present our result in detail in §3. However, we did not add the statistics for unlabeled posts in RQs, as these posts did not unveil any ML contracts.

**Table 2:** Dataset for Empirical Study on ML Contracts

Library	SO # Posts				# Contracts
	Criteria [1.]	Criteria [2.]	Criteria [3.]	Criteria [4.]	
<i>TensorFlow</i>	24368	2205	1400	605	162
<i>Scikit-learn</i>	12506	1641	1127	551	122
<i>Keras</i>	12300	1285	821	333	103
<i>PyTorch</i>	2500	439	313	76	26
Total	51674	5570	3661	1565	413

## 2.2 Filter Dataset

We processed the collected posts further to enable a classification scheme for contracts. We followed two main steps to filter these posts. The *initial step* is an *automatic* pre-processing of the collected posts based on the following criteria: [1.] Within these posts, authors asking the question must include some code snippet(s). We reason that a question post discussing these libraries and including snippets of code is more likely to have difficulty with API contracts,

<sup>1</sup> <https://stackoverflow.com/questions/42480111/>

thus may show challenges related to contract violation for relevant APIs. Furthermore, posts with code enable identifying the ML APIs being used. We have collected a total of 51674 posts with this filtering criteria. [2] We further filter the posts having a score higher than five based on the guidelines from prior works Nasehi et al. (2012); Zhang et al. (2018b); Islam et al. (2019, 2020) to ensure that posts are of high quality. This additional criterion produced a total of 5570 posts. [3] We considered posts with accepted answers (having a score higher than five) only, as those answer posts have successfully identified and resolved the problem faced by the author of the question post. The criterion for including accepted answers to the dataset follows prior studies Islam et al. (2019); Zhang et al. (2019); Islam et al. (2020) that have argued that if the answer of a post is not accepted, then that answer might not have addressed the issue. This step enabled us to collect 3661 posts in total. The steps up to this point are automatic. All queries to filter datasets according to this set of criteria are provided in the *figshare* repository<sup>2</sup>.

[4] Furthermore, the accepted answers frequently contain code, and we expect that these code snippets focus on the required contracts for ML APIs.

Additionally, we manually checked if the accepted answer to a post clearly describes the API contracts using text (without code). If this is true, we also consider such a post. The question posts in our study provided us with the contract violations ML software is susceptible to and the accepted answer posts directed us towards the contracts. Thus, the considered *SO* posts capture both contract violations and potential contracts. After these stages, we curated a total of 1565 posts; the posts specific to each of *TensorFlow*, *Scikit-learn*, *Keras*, *PyTorch* contained 605, 551, 333, and 76 posts (Table 2), respectively.

The posts obtained after manual filtering each contain at least one ML API-related contract but may contain more. Our study observed a blend of behavioral and temporal contracts for ML APIs. We called this a hybrid category in our classification (in §2.3). The posts from where we have extracted such contracts are the source behind multiple contracts from a single post. If multiple contracts were present in a single post, we extracted all contracts and labeled these using our taxonomy. For instance, in one *SO* post (*SO* post 6), it contained two contracts. The API in question is `random_shuffle()` from the *TensorFlow* library. The first extracted contract is to specify the argument seed with the desired value. The second extracted contract is to call `tf.random_shuffle()` and then call `tf.reset_default_graph()`. And the `random_shuffle()` API will ensure a shuffled *Tensor* if the contract is maintained in either of the ways mentioned.

Besides, the *SO* forum has a general strategy to tackle duplicate questions with the same (potential) answers. Users can flag a question on *SO* if it is analogous to a previously posted question concerning a concept. According to *SO*, the reasoning behind marking duplicate questions is that users should not discuss duplicate questions, but anyone with the same query can refer

<sup>2</sup> <https://figshare.com/s/c288c02598a417a434df>



to the previously posted discussions. In our study, we have found 359 unique contracts from a total of 413 contracts reported. SO’s strategy for flagging duplicating questions has enabled us to collect many unique ML contracts.

We note that the forum may contain other relevant ML API posts but not included in our dataset if the posts do not contain any contract or match the filtering criteria mentioned above. We have inspected the impact of imbalance in our dataset across libraries and address this in §3.5.

Next, we present a classification for ML API contracts and associated root causes in §2.3. This classification is used to identify and label posts with ML contracts. §2.5 demonstrates the taxonomy of the effects of these root causes of contract violations. Finally, we present a classification to identify locations of ML API contract violation (§2.4) based on ML pipeline stages.

### 2.3 Classification of ML Contracts and Violation Root Causes

To label the contracts for ML APIs found in our dataset, we developed a classification scheme that categorizes different types of contracts originating from these APIs.

**As mentioned earlier, the literature mainly discusses two types of contracts: behavioral and temporal.**

Typically, behavioral contracts for APIs consist of assertions that are required to be true before calling the API (preconditions) and assertions that must be valid upon exiting the API method (postconditions). In contrast, temporal contracts are those that capture the required order of API calls to ensure proper behavior. Both types of contracts are also observed in non-ML APIs, and we build our classification on top of this well-established classification. Building on an existing classification scheme helped us to not reinvent known ideas Glaser (1978) related to API contracts. Student authors in this work used open coding to build the extension appropriate for ML APIs.

**Process:** Researchers advocate using open coding to create any taxonomy Sarker et al. (2000); it is best that the researchers perform the task themselves rather than rely on a third party. The authors worked as a group initially to perform the coding and sampled 10% data to that purpose. This strategy had several advantages, e.g., a consistent decision to choose between existing concepts and create a new one; categories became more exact while differences became more evident than individually proposed taxonomy categories, and it also provided an opportunity to properly train the two labelers. We used *axial coding* Corbin and Strauss (2008), a technique that helps to collapse core themes involving qualitative data. In other words, it organizes the codes developed during open coding. This technique is used for cases where conceiving sub-categories seems necessary for any central component inside the classification schema. To elaborate, in our study, as we analyzed and labeled the SO posts with identified contracts, we looked at how these sub-categories could be grouped into central categories, so that the central category could encompass a number of different posts. In some cases, these central categories (axes)

are from the state-of-the-art taxonomy, e.g., data type-related contracts, but in other cases, a new abstract category seemed appropriate, e.g., selection. For instance, the codes such as *Primitive Type*, *Built-in Type*, etc., are well-established codes that describe different categories of type-related contracts. We used axial coding to identify that these contracts can be collapsed into the sub-core theme of checking *Data Type*-related contracts. Similarly, we organized sub-core core categories eventually into core categories. For instance, *Data Type* is organized under the core category *Single API Method*. We further use relational and variational sampling Corbin and Strauss (1990) using *SO* data to support or contradict the relationship between sub-categories and core categories. These sampling techniques facilitated explaining relations between theoretically relevant categories through gathering data (depending on the frequency of similarity or variation) on each group, e.g., considering conditions, consequences, etc., on a case-by-case basis. For example, we located instances of the leaf category *ML type* in our dataset that describes special type-related contracts that is only present in ML APIs. The multiple samples we collected indicated that the reason behind this contract violation is the *input of an unacceptable input type*, and the effect, if explicitly present in the samples, is *crash*. The frequency of such similarity confirmed the relationship between the category *ML type* and the category *Data Type*. This is an example of relational sampling, precisely.

**New Categories:** We found four new categories during our initial study (marked with ● in Table 3). After the initial study, the labelers individually studied the rest of the posts and were at liberty to suggest additional categories if the need arose (detail on labeling in §2.6). The labelers conducted an in-person meeting under the supervision of a moderator to discuss the suggested additional categories and these reconciliation effort resulted in one additional category (marked with ► in Table 3).

**Classification Scheme:** Next, we described our obtained classification schema in detail. Furthermore, all categories included in this classification are shown in Table 3. At the top level, we presented three central contract component levels: contracts involving *Single API Method*, contracts involving *API Method Order*, and contracts that required a *Hybrid* of preceding categories. The first fundamental category, *Single API Method (SAM)*, in our classification scheme captures preconditions/postconditions involving a single API method. This core category is based on behavioral contracts. Next, ML APIs often require particular call orderings to demonstrate normal behavior; we classify contracts specifying such order as **API Method Order (AMO)**. This category is based on temporal contracts. Subsequently, we classified these categories into sub-classes until we could find a leaf category that denoted the contract of a particular type for ML APIs. For each such class, we explained the root cause of that contract violation subsequently.

**Table 3:** Type of Contracts for ML APIs (Symbols ● and ▶ at the end of leaf components designate novel categories)

Level 1	Level 2	Level 3
Single API Method (SAM)	Data Type (DT)	Primitive Type (PT) Built-in Type (BIT) Reference Type (RT) ML Type (MT) ●
	Boolean Expression Type (BET)	Intra-argument Contract (IC-1) Inter-argument Contract (IC-2)
API Method Order (AMO)	Always (G) Eventually (F)	
Hybrid (H)	SAM-AMO Interdependency (SAI)	SAM (Level 3) $\wedge$ AMO(Level 2) ● SAM (Level 3) ▶
	Selection (SL)	AMO (Level 2) ● Comb. of SAM(Level 3) and AMO(Level 2) ●

\* **Green** cells indicates the behavioral contract. **Blue** denotes temporal contract and **Orange** cells indicate the hybrid respectively.

### 2.3.1 Type of Contracts involving Single API Method (SAM)

The first sub-category of Single API Method (SAM) contract concerns type checking that is required **Data Type (DT)** of API arguments.

This subclass consists of four types of contract:

**Primitive Type (PT):** This represents the ML API argument type can be a primitive type, e.g., `float`, `int`, `bool`, `number`, `None`, and the rest. For instance, in *SO* post 1, the `decode()` method from the *TensorFlow* library expects a `byte string`. The root cause of this contract violation is an input of an unacceptable type.

```
1 hello = tf.constant('Hello, TensorFlow!')
2 sess = tf.Session()
3 print(sess.run(hello))
```

#### ✓ Accepted Answer

Use `sess.run(hello).decode()` because it is a bytestring, decode method will return the string.

#### Stack Overflow post 1: Example post with contract

```
1 conv2 = conv2d(conv1, wts['conv2'])
2 conv2 = maxpool2d(conv2)
3 conv2 = tf.reshape(conv2, shape=[-1,158*117*64])
4 frame = tf.placeholder('float', [None, 640-10, 465, 3])
5 controls_at_each_frame = tf.placeholder('float', [None, 4])
6 conv2 = tf.concat(conv2, controls_at_each_frame, axis=1)
```

#### ✓ Accepted Answer

`conv2 = tf.concat((conv2, controls_at_each_frame), axis=1)` ; Note we need two frames that are required to concatenate within parentheses, as specified

#### Stack Overflow post 2: Example post with contract

**Built-in Type (BIT):** The contracts involving more complex built-in types (such as `dict`, `list`, `tuple`, and `array`). For example, in *SO* post 2, `concat()`

from the *TensorFlow* library expects the first argument to be of `array` type. The root cause of this contract violation is an input of unacceptable type.

**Reference Type (RT):** This category of contracts can involve either internal class object, i.e., referenced class objects within the API class, or external class object, i.e., external variable referenced from separate modules of the ML library. For example, in *SO* post 3, a contract for the API `KerasRegressor()` from *Keras* is shown. The argument accepts a function, an instance of a class that implements the call method or `None`. As the argument `build_fn` of this API accepts reference type as one of its expected argument types, we classify this under the reference type category. The root cause of this contract violation is that an input of unacceptable type is supplied to the method.

```
1 model = nn_model()
2 model = KerasRegressor(build_fn=model, nb_epoch=2)
```

#### ✓ Accepted Answer

Herein lies the issue. Rather than passing your `nn_model` function as the `build_fn`, you pass an actual instance of the Keras Sequential model. One of the following three values could be passed to `build_fn`:

- A function
- An instance of a class that implements the call method.
- None

**Stack Overflow post 3:** Example post with contract

**ML Type (MT):** This final contract component of data type contains ML types. ML types are a multidimensional array with a uniform type (`float16`, `float32`, `complex16`, etc.), particularly designed for ML pipelines to achieve accelerated performance (i.e., ease of use with GPU).

For instance, in *SO* post 4, an *ML Type* related contract is captured stating that the `matmul()` API from the *TensorFlow* library requires that both of the arguments should be a `Tensor` with one of the following types: `float16`, `float32`, `float64`, `int32`, `complex64`, `complex128`.

```
1 layer_1 = tf.nn.relu(tf.add(tf.matmul(_X, _weights['h1']), _biases['b1']))
2 layer_2 = tf.nn.relu(tf.add(tf.matmul(layer_1, _weights['h2']), _biases['b2'])))
3 return tf.matmul(layer_2, weights['out']) + biases['out']
```

#### ✓ Accepted Answer

The `tf.matmul()` op does not perform automatic type conversions, so both of its inputs must have the same element type. The error message you are seeing indicates that you have a call to `tf.matmul()` where the first argument has type `tf.float32`, and the second argument has type `tf.float64`. You must convert one of the inputs to match the other, for example using `tf.cast(x, tf.float32)`.

**Stack Overflow post 4:** Example post with contract

Another example of this type of contract 5 is that the first two arguments for the `fit()` API should have the type of a `numpy array` or a `list of numpy`

**arrays.** The root cause of this contract violation is an input of unacceptable type supplied to the method. This post also shows that the API has a supplementary contract concerning argument dependency, as discussed below.

```
padded_model.fit(train_X, train_y, epochs=50, verbose=1)
```

#### ✓ Accepted Answer

If `train_x` and `train_y` are normal *Python* lists, they don't have the attribute `.ndim`. Only Numpy arrays have this attribute representing the number of dimensions.

#### Stack Overflow post 5: Example post with contract

The API method can also involve Boolean assertions related to its argument values, **Boolean Expression Type (BET)**, instead of only type related checks. We classify these types of contracts into two subclasses:

**Intra-argument contracts (IC-1):** IC-1 specifies preconditions related to a single argument of the API. These contracts may involve both comparisons and logical combinations.

```
a_shuf = tf.random_shuffle(a)
```

#### ✓ Accepted Answer

That only sets the graph-level random seed. If you execute this snippet several times in a row, the graph will change, and two shuffle statements will get different operation-level seeds. To get deterministic `a_shuf` you can either

- Call `tf.reset_default_graph()` between invocations or
- Set operation-level seed for shuffle: `a_shuf = tf.random_shuffle(a, seed=42)`

#### Stack Overflow post 6: Example post with contract

An example of an IC-1 contract is given in *SO* post 6, which shows an ML API users trying to use the *TensorFlow* API `random_shuffle()` to shuffle a Tensor, `a`, with some set seed value. One of the solutions mentioned in the accepted answer says that to do that, one should specify the argument `seed` with the desired value, e.g., the argument `seed` gets the value 42. The root cause of this contract violation is that acceptable input value is not supplied to (the `random_shuffle()`) method.

**Inter-argument contracts (IC-2):** IC-2 contracts involve more than one argument to an API method, possibly using comparisons or logical expressions. For example, in *SO* post 4, the `matmul()` API from *TensorFlow* requires that the type of the second argument should match the type of the first argument. A comparison expression can express this contract, so it belongs to IC-2. The root cause of this contract violation is that the (`matmul()`) API is missing input value/type dependency between arguments. Another example 7 for this category is `nn.softmax_cross_entropy_with_logits()`, an API from *TensorFlow*, which requires that the logits and labels arguments must have

the same shape (i.e., `[batch_size, num_classes]`).

```
1 tf.nn.softmax_cross_entropy_with_logits(
2     logits=b, labels=a).eval(feed_dict={b:np.array([[0.45]]), a:np.array(
3         [[0.2]])})
```

#### ✓ Accepted Answer

Like they say, you can't spell "softmax\_cross\_entropy\_with\_logits" without "softmax". Softmax of [0.45] is [1], and log(1) is 0. logits and labels must have the same shape `[batch_size, num_classes]` and the same dtype (either float16, float32, or float64).

**Stack Overflow post 7:** Example post with contract

### 2.3.2 Type of Contracts involving API Method Order (AMO)

Multiple APIs can be involved in an AMO contract. There are two sub-categories as follows:

**Always (G):** Always contracts are AMO contracts that hold at each point of history. For example, as shown in *SO* post 8, for *TensorFlow*, the call to the method, `tf.wholeFileReader()` must be followed by another method call, `tf.train.start_queue_runners()` to avoid hanging. The root cause of this contract violation is that the *always* required order between these calls is not followed.

```
1 def read_image(filename_queue):
2     reader = tf.WholeFileReader()
3     key,value = reader.read(filename_queue)
```

#### ✓ Accepted Answer

If you're not using feeding—i.e. using the `tf.WholeFileReader` as shown in your program—you will need to call `tf.train.start_queue_runners()` to get started. Otherwise your program will hang, waiting for input.

**Stack Overflow post 8:** Example post with contract

```
1 model = Sequential()
2 model.add(LSTM(100, input_dim = num_features))
3 model.add(Dense(1, activation='sigmoid'))
```

#### ✓ Accepted Answer

The solution to this: you need to enable the LSTM layer to return a sequence instead of only the last element. Since the `Dense` layer is not able to handle sequential data you need to apply it to each sequence element individually which is done by wrapping it in a `TimeDistributed` wrapper.

**Stack Overflow post 9:** Example post with contract

**Eventually (F):** Eventually contracts are AMO contracts where the ordering is only required at some point in history. In other words, this specifies that a required API ordering must be true at some point in this program's execution history far enough in the future. For instance, in *SO* post 9, the author is trying to solve a sequential classification (input data where order matters) task. In the model, they used the `LSTM()` API to return a sequence and then output it as a `Dense()` object. The activation function has a one-to-one correspondence with the type of classification being performed. For that reason, the `sigmoid` function is rightly used. However, in the model, they used the `LSTM()` API to return a sequence and then output it as a `Dense()` object. This method order of APIs demonstrates an incorrect API method order, as the order in the question post is missing a `TimeDistributed()` API call.

Note that the code is correct for a many-to-one task in natural language processing (NLP). However, in this question, the user asks for a many-to-many solution, in which case it becomes mandatory to apply `TimeDistributed()`. Therefore, using the `TimeDistributed()` API only becomes a requirement after the `LSTM()` API is used to return a sequence. The root cause of this contract violation is that in a state where a call to a method (`LSTM()`) returns (a sequence), another call to a method (`TimeDistributed()`) should have occurred. Thus, in this *SO* post 9, this *eventually* contract is violated because the author did not know that the `TimeDistributed()` API is a requirement to be called eventually after the `LSTM()` API is used to *return a sequence*.

### 2.3.3 Type of Contracts involving Hybrid (H) of SAM and AMO

The **Hybrid (H)** category involves a blend of behavioral and temporal contracts. This category has two subclasses:

```
clf = GridSearchCV(SVC(C=1), tuned_parameters, score_func=auc_score)
```

#### ✓ Accepted Answer

As noted already, for SVM-based Classifiers (as `y == np.int*`) **preprocessing** is a must, otherwise the ML-Estimator's prediction capability is lost right by skewed features' influence onto a decision *[sic]* function.

**Stack Overflow post 10:** Example post with contract

**SAM-AMO Inter-dependency (SAI):** SAI contracts have a dependency between behavior and method orders. This dependency could be in either direction, i.e., the program's state could determine the order of API calls, or the order of API calls could require that some condition must hold. For example, in *SO* post 10, if an ML API users uses the SVM-based classifier `SVC` as the *estimator* parameter for `GridSearchCV()` with *Scikit-learn*, then `preprocessing.scale()` must precede this call. Since the order of the method

calls `GridSearchCV()` and `preprocessing.scale()` APIs is dependent upon the value given to the parameter of `GridSearchCV()`, it belongs to the *SAI* contract category. The root cause of this contract violation is that the value being passed to one method call (`GridSearchCV()`) requires a temporal ordering between the two (`GridSearchCV()` and `preprocessing.scale()`) methods.

The leaf components of this subclass contain all contract cases that we derived individually for the *SAM* and *AMO* categories. For example, if an *intra-argument contract*, *IC-1* of an API determines an *always (G)*, order of two APIs like the example above then, it belongs to **SAM (Level 3)  $\wedge$  AMO (Level 2)**.

Any dependency between SAM related leaf nodes, e.g., primitive type, built-in type, ML type, etc. and AMO related leaf nodes, i.e., G and F, belong to this category.

**Selection (SL):** The final subclasses in our classification are those contracts that involve a choice when it comes to enforcing an API related contract. If the choices only belong to the contract components of *SAM* or *AMO*, then we categorize the contracts into either **SAM (Level 3)** or **AMO (Level 2)**, respectively. For instance, in *SO* post 11, the author wants to convert two `numpy` arrays to `Tensors` and uses `TensorDataset()` from the *PyTorch* library. The arguments of this API must either be of `Double` or `Float` type. The API then confirms `DoubleTensor` conversion upon exiting. Hence, there are two choices of category SAM (specifically IC-2) to maintain the contract for this API, and we mark this with **SAM (Level 3) category**. The root cause of this contract violation is that the client did not follow one of the two choices (providing arguments of `Double` or `Float` types).

```
train = data_utils.TensorDataset(torch.from_numpy(X).double(), torch.
    from_numpy(Y))
train_loader = data_utils.DataLoader(train, batch_size=50, shuffle=True)
for batch_idx, (data, target) in enumerate(train_loader):
    data, target = Variable(data), Variable(target)
    optimizer.zero_grad()
    output = model(data)                                # error occurs here
```

#### ✓ Accepted Answer

The `numpy` arrays are 64-bit floating point and will be converted to `torch.DoubleTensor` standardly. Now, if we use them with our model, we'll need to make sure that your model parameters are also `Double`. Or we need to make sure, that your `numpy` arrays are cast as `Float`, because model parameters are standardly cast as `float`.

#### Stack Overflow post 11: Example post with contract

Another example can be seen in the *SO* post 12, where the author of the post is using the *Keras* library to create a neural network. Then they want to initialize and fit the neural network weights and save these weights. Next, they want to use these saved weights and predict some output values given the inputs. However, they had issues using the `load_weights()` API to collect the saved weights. The answer post explains that as one uses the `load_weights()` API, one has to maintain an order between two other related APIs (`compile`



and `predict()`). One expected order is calling `load_weights()`, `compile()`, `predict()`. The order alternative is calling `compile()`, `load_weights()`, and `predict()` at some point in history. As both choices involve AMO, this belongs to the **AMO (Level 2)** category. The root cause of this contract violation is that the client did not make one of the two choices (maintaining the method order between the related APIs).

```
model = Sequential()
...
model.load_weights('keras_w')
y_pred = model.predict(X_nn)
```

#### ✓ Accepted Answer

We need to call `model.compile`. This can be done either before or after the `model.load_weights` call but must be after the model architecture is specified and before the `model.predict` call.

#### Stack Overflow post 12: Example post with contract

In comparison, if the choices involve both SAM and AMO, then we categorize the contract as a combination type contract **Comb. of SAM and AMO**. For example, in *SO* post 6, we observe such a combination. The accepted answer respondent mentions two alternative ways to maintain correctness when using the `tf.random_shuffle` API. The first choice is setting the argument `seed` for this API to some desired value. The second is maintaining an order between invocations of `tf.random_shuffle()` and `tf.reset_default_graph()`. Since the same contract breach can be resolved through either a behavioral or temporal contract that involves `tf.random_shuffle()` API, there is a selection involved as to which one to be adopted. Documentation should include all choices to maintain contracts for an API method. The root cause of this contract violation is that the client did not make one of the two choices (providing an acceptable `seed` value or using an acceptable method ordering) for the API to function properly. Researchers should emphasize the need to be able to express such requirements to users, who can choose to satisfy the requirements of a library either by maintaining a temporal order or by some state-based change. Therefore, the practitioners can design and develop a contract checking mechanism for ML API calling orders to facilitate the end-users.

## 2.4 Classification of ML Contract Violation Locations

As we investigated the requirements for ML contracts, we also classified the API locations of the contracts being violated.

We based this classification on prior works Guo, Yufeng (2017); Islam et al. (2019), and used a similar open coding strategy as we did when conceiving the classification for contract types. The categories are explained in Table 4.

**Table 4:** Contract Violation Locations

Data Preprocessing	These APIs pre-process data before feeding it to ML models.
Model Construction	These APIs are used to build ML models, either from scratch, by accessing a predefined model, or by compiling constructed models.
Model Evaluation	APIs used to estimate the generalization accuracy of a model.
Model Initialization	APIs used for initializing a predefined model, e.g., an API to load random weights for a model.
Train	Describes APIs that determine values for weights and biases of a model.
Prediction	Describes APIs that predict an outcome after training.
Hyper-parameter Tuning	Describes APIs that change hyper-parameter(s) that control the learning process.
Load	Describes APIs that load or store data from external storage.

## 2.5 Classification of Effects

We used a prior work Islam et al. (2019) for classifying the effects to the root causes discussed in §2.3. It has six categories: bad performance (BP), crash (C), data corruption (DC), hang (H), incorrect functionality (IF), and memory out of bound (MOB). We have added one category Unknown (U) besides these categories to identify cases that remain non-classified. The details about this classification of effect are discussed in Table 5.

**Table 5:** Contract Violation Effects

Bad Performance	Common effect in ML software; ML API users face model problems even though they use deep learning APIs correctly because APIs in these libraries are abstract.
Crash	Frequent effect in ML. In fact, any kind of contract violation can lead to a Crash. A symptom of the crash is that the software stops running and prints out an error message.
Data Corruption	This happens when the data is corrupted while flowing through the network, and a user gets unexpected output. This effect is a consequence of misunderstanding the ML APIs.
Hang	Hang is caused when ML software ceases to respond to inputs due to slow hardware or inappropriate ML algorithm. Software running for a long period of time without providing the desired output is considered as a symptom.
Incorrect Functionality	It occurs when the software behaves unexpectedly without any runtime or compile-time error due to the incorrect output format, model layers not working desirably, etc.
Memory Out of Bound	ML software often halts due to the unavailability of the memory resources for the wrong model structure or not having enough computing resources to train a model.
Unknown	Sometimes the effect of ML contract violation is unknown. We have added this category for cases that remain non-classified.

## 2.6 Labeling

The classification schemes described in §2.3, §2.4, and §2.5 were used to label all 1565 collected *SO* posts. First, the second and the third authors with strong ML background, have familiarized themselves with contract literature. The authors have all studied key papers in the area of software specification and design-by-contract methods. Then we trained these two authors to understand the classification schema with the help of some example posts. In this training process, the two authors were shown multiple examples for each category in the classification schema. The examples were demonstrative of where the contract is broken for an ML API and how the accepted answer describes the correct usage for that precise API. Then, each rater performed independent labeling of these posts in two iterative rounds. The 10% sampled data analyzed for the classification coding scheme and the first iterative round of labeling served as part of the training process for the labelers. To measure the inter-rater agreement, we have used Cohen’s Kappa coefficient Viera et al. (2005) as labeling progressed at 1%, 2%, 5%, 10%, and continued in this fashion. We have followed the methodology used in prior works Höst et al. (2005); Chatterjee et al. (2020); Islam et al. (2019, 2020) to reconcile inter-rater disagreements at fixed intervals. During first iterative round, at 5% and 10%, we report the Kappa coefficient to be 40% and 51%, respectively. The low value of the agreement directed the raters to meet more frequently (at each 2%) for a second iterative round during the first few intervals to clarify the labels that raters were using for each post. During these meetings, raters discussed the reasoning behind cases where a strong disagreement occurred in a moderator’s presence. We continuously checked the Kappa coefficient at these intervals, and even if the Kappa value fluctuated we reached values over 80% after completing labeling 22% of the posts for the entire dataset. According to Sim and Wright (2005), a Kappa coefficient value higher than 0.80 is considered as almost perfect agreement.

## 3 Results

The main question we asked is about what contracts are most needed by ML API users. In this section, we present the quantitative data (e.g., representing contract violation patterns, root cause, effect, contract comprehension challenges, etc.) to show the places where immediate support for contracts is needed. Hence, we analyze the results from our *SO* study to answer the research questions from §1, report our findings on the original (and the filtered subset) dataset described in §2.1, and discuss implications and actionable insights.

### 3.1 Contract Frequency, Root Cause, and Effect of Contract Violations

In this subsection, we answer **RQ1** by presenting the required types of ML contracts, the root causes of contract violations, and related effects.

#### 3.1.1 Required ML contracts and associated root causes

To explore required ML contracts and the root causes behind contract violations, we use the leaf contract types from our classification (§2.3) schema. Table 6 shows the frequency of each type of contract from the classification found in our dataset. Figure 3 demonstrates the corresponding root causes. Figure 2 shows the statistical comparison of ML Contracts for two datasets (all filtered posts and the subset containing posts with scores of 30 or higher).



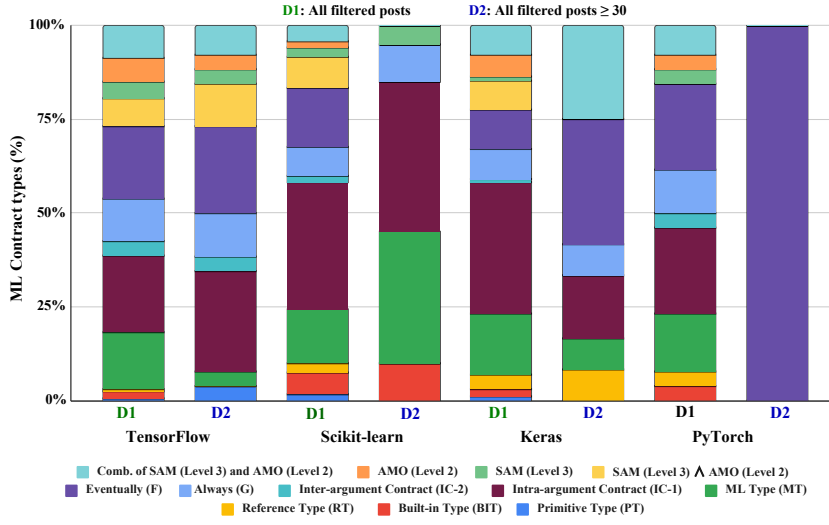
**Finding 1** : Most frequent ML API contracts are:

- constraint check on single arguments of an API.
- order of APIs that become a requirement eventually.

**Table 6:** Statistics of ML Contracts in *SO*

Contract Types	ML Library				Overall
	<i>TensorFlow</i>	<i>Scikit-learn</i>	<i>Keras</i>	<i>PyTorch</i>	
Primitive Type (PT)	0.63%	1.65%	0.97%	0.00%	0.01%
Built-in Type (BIT)	1.88%	5.79%	1.94%	3.85%	3.18%
Reference Type (RT)	0.63%	2.48%	3.88%	3.85%	2.20%
ML Type (MT)	15.00%	14.05%	16.50%	15.38%	15.16%
Intra-argument Contract (IC-1)	20.63%	33.88%	34.95%	23.08%	28.36%
Inter-argument Contract (IC-2)	3.75%	1.65%	0.97%	3.85%	2.44%
Always (G)	11.25%	7.44%	7.77%	11.54%	9.29%
Eventually (F)	19.38%	15.70%	10.68%	23.08%	16.38%
SAM (Level 3) $\wedge$ AMO (Level 2)	7.50%	8.26%	7.77%	0.00%	7.33%
SAM (Level 3)	4.38%	2.48%	0.97%	3.85%	2.93%
AMO (Level 2)	6.25%	1.65%	5.83%	3.85%	4.65%
Comb. of SAM (Level 3) and AMO (Level 2)	8.75%	4.13%	7.77%	7.69%	7.09%

**Required ML Contract.** We identify that breaking the contract on the *single argument of an API (IC-1)* and *eventually (F) required API method orders* are the most frequent type of contracts violated. We observe that the lack of domain knowledge, and incomplete error messages are some of the reasons why ML API users struggle with the IC-1 category. For example, in *SO* post 6 the author struggled to grasp the difference between graph level seed and operation level seed when using the `tf.random_shuffle` API. In addition, some ML APIs are involved in *AMO* contracts that require particular method orders. This required method order is often a source of confusion. For the posts with score  $\geq 30$  in *PyTorch* library (2), all observed contracts belong to *AMO* category. However, the number of posts with a score of 30 and higher and containing a contract from the *PyTorch* library is very low (3 contracts). Thus, we refrain from making any additional observations for this case. To



**Fig. 2:** Comparison of ML Contract types of all filtered posts (D1) and subset with score  $\geq 30$  (D2) in ML libraries

analyze further why the required contracts mentioned in this finding are commonly violated, we have randomly sampled ML APIs from our dataset and studied the documentation for these APIs to investigate if the documentation is complete. We have analyzed API documentation from the *Keras* and *TensorFlow* libraries and observed that many of these incorrect usages of APIs are not documented, especially the corner cases. As an instance, the function `ReLU` is a valid activation function for ML layer APIs in *TensorFlow*. However, it should not be used if the layer API in question is the output layer of the model in a multi-label classification.

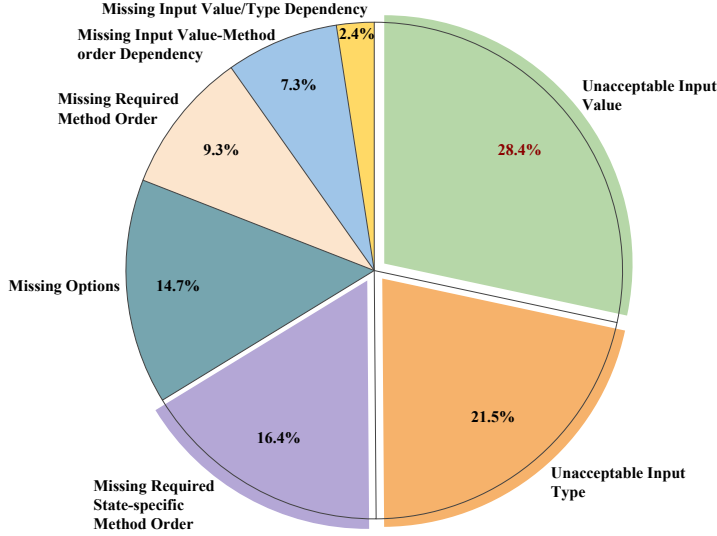
The SE community can employ existing contract mining approaches Zhong et al. (2020); Reger et al. (2013); Lemieux et al. (2015); Lemieux (2015); Le and Lo (2018) to mine these contracts and enhance library documentation.



**Finding 2 :** ML APIs require ML type checking contracts and show inter-dependency between behavioral and temporal contracts.

We have observed ML type checking (MT) is the next major category, considering all posts. For instance, in one *SO* post<sup>3</sup>, the ML API users is trying to use a predefined model through a *TensorFlow* API `seq2seq()`. This API essentially consists of two recurrent neural networks. The encoder part processes the input and the decoder generates the output. To capture this, `seq2seq()` contains two arguments `encode_inputs` and `decode_inputs`. The contract requirement for these arguments according to the accepted answer is that if the

<sup>3</sup> <https://stackoverflow.com/questions/33762831/>



**Fig. 3:** Distribution of root causes behind ML contract violations

input has some shape `[n]`, then both of the arguments are required to have a shape of `[batch_size × n]`. We also note that, the ML-type checking error is more common (for the posts with score  $\geq 30$ ) in the *Scikit-learn* library compared to other libraries. This is one of the key findings that is different when we compared the original curated dataset and the filtered dataset with posts scored 30 or higher. This observation can be attributed to the fact that the other studied ML libraries incorporate some type checking system, unlike *Scikit-learn*. As a result, a *TensorFlow* or a *Keras* or a *PyTorch* program is less likely to contain type errors. For instance, we have described that in the *SO* post 4, the `matmul()` API from the *TensorFlow* library requires that both of the arguments assume that the same `Tensor` types will be provided by the caller of the API. Therefore, supplying anything other than the allowed type will cause a type error and the program to crash. In contrast, *Scikit-learn* does not require its program to be strongly typed and relies on Python’s default type system. This situation highlights the need for type regulation in the ML framework. A runtime assertion checking tool could help catch such contract violations; such a tool could be built, for example, by enhancing an off-the-shelf (e.g., PyContracts Graham et al. (2010)) tool that can detect violations of the ML-type contracts we propose. We note that some of the type issues may be caused by the dynamically typed nature of the programming language, Python, and are out of the scope for this paper.

Additionally, we see that one other new category (dependency between behavioral and temporal contracts) in our classification is required for significant number of APIs. Contract languages and type checking tools Lehtosalo (2012); Seshia et al. (2018); Jothimurugan et al. (2019) should add sufficient expressiveness for these additional types of contracts seen in ML APIs.

We note that the behavioral contracts reported in this study are largely preconditions. However, we found some postconditions as well. For instance, in SO post<sup>4</sup>, the author is using the `tensorflow.session()` method to return a `Session` object. A `Session` is a class that is used to run *TensorFlow* operations. Then calling the `run()` method on this `Session` object allows evaluation of the `Tensor`. In the example post cited above, the `Tensor` is a constant `String`. It is supplied as the value used for the argument `Fetches`. We know that any value in a `Tensor` holds the same data type with a known (or partially known) shape. In this example, the value returned by `run()` has the same shape as the `Fetches` argument. Now one can decode this output data as needed. The contract we see in this SO post is a postcondition. The contract for the `tf.io.decode_raw()` API is "returns a binary string (python 2), byte string (python 3)" upon exiting the API call.



**Finding 3 :** Unacceptable input value is the most common root cause.

**Primary Root Cause.** We identify that supplying unacceptable input values to APIs is the primary root cause behind contract violation in ML. The ML API users fail to recognize acceptable input values often for several reasons, e.g., misunderstanding a hyper-parameter setting. The undesired input values found in our study can be utilized as test cases in ML systems and avoid some of these contract breaches.

### 3.1.2 Effects of Contract Violations

To realize the effects of the contract violations, we have used the classification of effects from a prior work Islam et al. (2019) mentioned in §2.5. Figure 4 illustrates the distribution of contract violation effects across libraries.



**Finding 4 :** On average, 56.93% of the contract violations for the ML libraries leads to a crash.

**Crash (C).** The majority of contract violations for ML APIs lead to a crash in the software and we observe the range within 42.31%-66.02%. This result varies only for *PyTorch* with post score  $\geq 30$ . *Scikit-learn* has the most examples of contract violations that have lead to crashes among the chosen libraries. As an instance, in SO post 4, violating the inter-argument contract (IC-2), would result in a crash for the program. Researchers might build a new automated repair tool inspired by existing repair tools Le Goues et al. (2012); Mechtaev et al. (2016); Pei et al. (2014); Long and Rinard (2015). In this regard, mined contracts could be utilized that lead to crashes and act as a preemptive measure for the code.

<sup>4</sup> <https://stackoverflow.com/questions/40904979/>

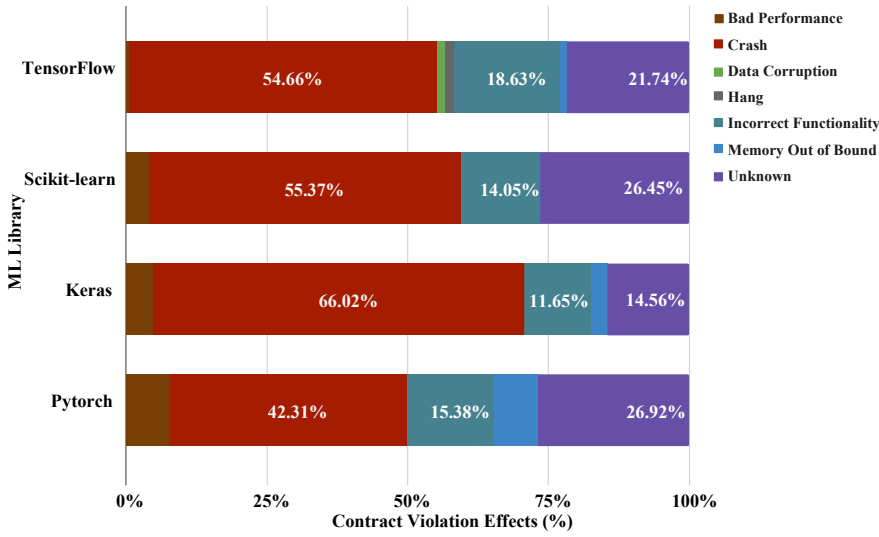


Fig. 4: Distribution of ML contract violation effects



**Finding 5 :** Incorrect functionality has a different frequency pattern in *Keras*.

**Incorrect Functionality (IF).** The next frequent (15.33% on average) effect that we observed in terms of contract violation is causing incorrect functionality in ML software. ML API users apprehend the deviant behavior either from experience or through the logical organization of the ML model.

However, the library *Keras* shows a lower percentage. This divergence in frequency can be explained by the fact that *Keras* is a high-level deep learning library; thereby hiding many complicated implementation details which reduces the chance of running into IF. Since the compiler cannot catch incorrect functionality, our dataset can become a benchmark through contract annotation to detect this effect. Expert ML API users can further rank these particular contract violations (e.g., scary, troubling, concern) to provide a hint of severity for these IFs as many bug detection tools (such as Bugram Wang et al. (2016), Salento Murali et al. (2017)) do.



**Finding 6 :** Contract violation-effect distribution is similar across libraries.

**Contract violation-effect correlation.** To determine the effect of breaking a contract, we used the information explicitly available in the *SO* post we labeled. Even so, we hypothesize that by observing the type of contract violation, it is possible to make an informed guess on the corresponding effect. For instance, Islam *et al.* reported that the violation of type or shape usually



results in a crash during runtime Islam et al. (2019). One *SO* post <sup>5</sup> author was not sure how to use the API `DataLoader()` from the *PyTorch* library. The answer post lists that the API in question requires that the argument type should be a subclass of `Dataset` class. Even though it was not explicitly described on the post, such a type checking contract violation would result in a crash. *Python* being a loosely typed language, type mismatch may go unnoticed during compilation. It usually crashes for mismatches in the expected type or shapes Islam et al. (2019).

To test this hypothesis on learning from other ML libraries regarding violation-effect correlation, we obtained the conditional probabilities,

$$\Pr(E = \text{effect}_i \mid V = \text{violation}_j)$$

which describe how likely a certain effect ( $\text{effect}_i$ ) follow given a contract violation ( $\text{violation}_j$ ). Then, we utilized the *Jensen-Shannon divergence (JSD)* Endres and Schindelin (2003) measure to compute the distance between two probability distributions,  $E$  and  $V$ . The divergence ranges from 0 to 1, where 0 indicates perfect similarity, and 1 indicates no similarity.

In our experiment, we observed that the violation-wise effect distribution is similar across chosen libraries. The result shows that eventually (F) required method order, ML type checking (MT), and intra-argument contracts (IC-1) demonstrated a divergence score of 0.08, 0.11, and 0.14, respectively, with 10% support, indicating a similar effect distribution across libraries. This experiment agrees with our hypothesis. Therefore, the SE community can learn from contract violations of the same category for ML libraries and estimate unexpected behaviors of other ML libraries with similar effects in code. Furthermore, this experiment also shows an application of the proposed classification schema.



**Finding 7 :** Error messages thrown for breaching an ML contract are not often adequate at present.

**Error message.** In case of system failure, the crash or error message helps API users debug the code and identify the root cause. In the *SO* post 13, the author had received the error message in the listing below when they tried to load weight on a predefined model. It could be an exhausting task to understand the problem by only examining the error message. The answer to this post registers that the error occurs as the ML API users missed redefining the model architecture before loading weights. We find an error message inadequate if the error message is present in the author's post, and the response demonstrates that the error message presented does not reflect the incorrect usage for the API in question. Additionally, since domain experts can explain these challenging ML contracts (see §3.3), such extracted contracts can be encoded in a contract-checking tool. Such a tool, as a result,

<sup>5</sup> <https://stackoverflow.com/questions/44429199/>

can enable improved debugging mechanisms for ML software developers.

```

IndexError      Traceback (most recent call last)
<ipython-input-101-ec968f9e95c5> in <module>()
      1 model12 = Sequential()
--> 2 model12.load_weights("/Users/Desktop/SquareSpace/weights.hdf5") ...
IndexError: list index out of range

```

**Stack Overflow post 13:** Example post demonstrating inadequate error message

In our study, we found only a handful of contract violations that require runtime checks. For example, if overfitting happens during training, *regularization-related* APIs are necessary for the ML model stack. Additionally, we have found cases where runtime checks against the state alone are insufficient without more context. For example, in *Keras*, it is required to call `BatchNormalization()` between the *linear* and *non-linear* layer APIs in the model to achieve better performance. Thus, in this case, the presence of temporal history and an assertion check is required. For such kinds of ML contracts, we can extend the traditional design-by-contract approach Meyer (1992) to assert those contracts during runtime and utilize the contract violation message accordingly to inform the correct usage of ML APIs to the users.

In summary, we observed that the majority of the ML contracts are similar to traditional contracts, and Finding 1 indicates this. The contracts involving ML type checking and dependency between behavioral and temporal contracts are specific and needed by ML software. Interestingly, we report there are contracts that can be formalized as in traditional contracts, however, the contract violation effect is often different, e.g., bad performance, incorrect functionality, etc.; i.e., issues about performance and accuracy are more common in ML software.

### 3.2 Common Patterns for Contracts

This section highlights common patterns of contracts in the dataset, i.e., we analyze the common patterns of ML contract violations observed in our study. In section §3.1, we noted that IC-1, F, MT are the most frequently occurring patterns across libraries. These contracts are *atomic* in the sense that there is no dependency between behavioral and temporal contracts in these. We further investigated more complex contract patterns, including combinations of two or more atomic contracts, when answering **RQ2**. These types of patterns belong to the high-level category *hybrid* in our classification schema. Recall that hybrid contracts contain combinations, choices, or dependencies between the behavioral and temporal contracts.



**Finding 8 :** Eventually (F) related hybrid contracts are one of the most common patterns across ML libraries.

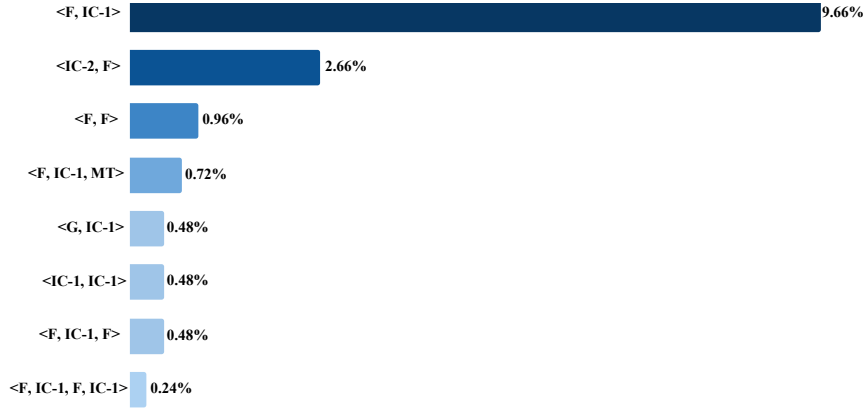


Fig. 5: Patterns of ML Contracts

**Patterns involving <F>.** Our result shows that F contracts (about the method orderings at a certain point in history) are the places ML API users struggle most, compared to G (always orderings); see Figure 5. For instance, we described that in *SO* post 10, the parameter choice for the `GridsearchCV()` API dictates whether it must be preceded by `preprocessing.scale()` API. In contrast, it is not always obvious for patterns involving F, e.g., ML API users sometimes use a pooling layer after a convolution layer to downsample the feature collected in the previous layer. Thus, this order is not mandatory for all program points. But, if the order is used<sup>6</sup>, then the API user should make sure the parameter `strides` of `tf.nn.conv2d()` is compatible with the `ksize` and `strides` parameters of the pooling layer (e.g., `tf.nn.max_pool()`) in *TensorFlow*. Violations of such hybrid patterns can be found using unit tests that capture variants of these patterns and testers should be aware of capturing these variants.

### 3.3 Difficulty in Contract Comprehension

In this section, we analyzed when the challenges are observed behind understanding ML contracts. Discovering categories of ML contracts is challenging because a significant number of *SO* posts do not contain contracts with accepted answers. Furthermore, the *SO* reputation of the user giving an answer does not necessarily determine whether an answer reveals a challenging ML contract. To discover the correlation between ML contracts and user expertise in terms of the contract violation issue and to answer **RQ3**, we have conducted an experiment that discerns respondent reliability and resolving time. We gather evidence from various perspectives described below to develop an

<sup>6</sup> <https://stackoverflow.com/questions/34092850/>

educated guess. Prior work Zhang et al. (2019) analyzes the scores of *SO* post to comprehend the types of deep learning questions are more difficult.

Inspired by that study, we have leveraged the *SO* reputation score to determine the overall expertise of a user. In our study, we slightly adapted the *SO* reputation score, and named it the *reliability score*. The reputation metric is often used to measure a user’s expertise level on *SO*, because it summarizes the overall impression of that user’s *SO* activity. We observe that a user can earn a reputation for various topics unrelated to ML-related skill sets. As a result, this metric poses a significant threat when we want to assess the expertise of a user in resolving ML contracts specifically. The *Reliability score* tries to mitigate this threat to an extent. Moreover, our adapted metric incorporates the number of accepted answers into account for higher confidence in the metric.

As an example, let us say, we are interested to know if a particular user from an *SO* post<sup>7</sup> is an expert in *Keras*. While the user has a high reputation score (26,164), they only have a score of 6 when filtered through the *Keras* tag from answering two questions. This indicates that this user has accumulated most of his reputation from other areas. So, to measure the expertise level of a user, we consider their score only on relevant tags. Since we have only included posts having *accepted* answers for this study, we refined this score to prioritize users having more accepted answers, which we call their *reliability score*, measured as follows:

$$reliabilityScore = totalScore \times \frac{(totalAcceptedAnswer + \mathbb{C})}{(totalAnswer + \mathbb{C})}.$$

Since a user may have no accepted answers, which would reduce their reliability score to 0; we add an equal constant value,  $\mathbb{C} > 0$  to both numerator and denominator of accepted answer percentage of the reliability score to prioritize among authors who do not have accepted answers. Here, we have used 1 as the value of  $\mathbb{C}$  in the study. For example, suppose two users *A* and *B* have obtained a total score of 1200 and 80 respectively by answering an equal number of questions without any accepted answers. In this case, the reliability score would be zero, had it not been for the normalization constant, and both the authors would be rated equally. As author *A* has achieved a significantly higher score compared to author *B* for the same number of questions answered, adding  $\mathbb{C}$  to the accepted answer fraction adds priority to the author with a higher answer score.

The dataset includes *average resolve time* for each type of contract, considering the time required to get accepted answers from the study and *reliability score* for these respondents.

We fitted the dataset in a linear regression model first. However, it violated multiple assumptions such as linearity and normality assumptions of residuals of linear regression. Therefore, we choose the kernel ridge regression technique, a non-parametric (without any underlying assumptions about distributions of

<sup>7</sup> <https://stackoverflow.com/users/5098368/>

**Table 7:** Expected reliability score of respondents to comprehend different contracts

Library	ML Contract (Leaf)	Reliability Score	Average Resolve Time (h)	Average Elapsed Time (h)	First Answer Accepted (%)
<i>TensorFlow</i>	IC-1	5.26	265.33	238.62	79
	MT	5.76	71.03	14.55	64
	SAM(Level3) $\wedge$ AMO(Level2)	4.51	136.65	20.25	67
	F	6.38	821.93	519.05	77
	G	10.16	320.48	174.77	61
<i>Scikit-learn</i>	IC-1	8.16	562.52	503.27	71
	MT	10.88	788.50	252.18	59
	F	8.15	47.85	0.28	71
<i>Keras</i>	IC-1	4.68	235.23	3.65	86
	MT	5.18	19.88	11.10	82
	F	8.20	1079.88	513.60	62
<i>PyTorch</i>	IC-1	6.60	84.38	0.00	100
	MT	4.83	25.45	0.00	100
	F	8.77	97.58	0.00	83

the dataset), and non-linear technique Murphy (2012). We used radial basis function (RBF) as a kernel for fitting nonlinearity of the dataset and a gamma value of 0.1 chosen through trial-and-error analysis. Since, in our dataset, we only included the accepted answers, the regression model predicts a minimum expected level of a user to solve the problem successfully. To that end, we use leaf contracts as features and the *reliability scores* as a target variable. Considering that features are categorical data, we converted them into a one-hot encoded vector to feed into the model. Table 7 shows the expected *reliability scores* and *average resolve time* for a *SO* post respondent to comprehend different contracts for all ML libraries. For example, to respond to an intra-argument contract (IC-1) from the *TensorFlow* library, a respondent’s expected *reliability score* is 5.26, and the *average resolve time* is 265.33 hours. Additionally, *reliability scores* are comparable for respondents within a library. Contract components with a support of less than 10% are excluded from consideration.



**Finding 9 :** For ML libraries, F contracts require a higher level of expertise and a longer average time to resolve.

A general observation is that *F* contracts have respondents with comparatively higher reliability scores, ranging from 6.38 to 8.77, compared to other types of contracts. Consequently, the average resolve time for these ranges from 47 to 1080 hours (approximately). We reason that this difficulty is because *F* contracts are not as evident as *G* contracts, since *F* contracts must only eventually hold before the program terminates. From our dataset, it is possible to provide a benchmark for experts who can resolve *F* contract violations. This benchmark could be used by the *SO* forum, for example, to recommend new ML-related posts to specific experts. Furthermore, *F* contracts often rely on an implicit assumption; a significant research direction could be automating ML program repair tools such as DLFix Li et al. (2020) to resolve this contract violation.



**Finding 10 :** For Scikit-learn, ML API users mostly struggle with comprehending ML type checking.

Surprisingly, we found that for *Scikit-learn*, ML API users mostly struggle with type checking contracts. The reliability score for this case is 10.88, and the average resolve time is 788.50 hours. We realize that *Scikit-learn* provides off-the-shelf ML algorithms for supervised and unsupervised learning, whereas, the other DNN libraries we have chosen allow API users to implement these deep learning algorithms and neural networks. Therefore, deep neural network (DNN) ML API users have some level of expertise towards ML type checking compared to the API users who use higher-level ML libraries such as *Scikit-learn*. Additionally, the DNN libraries in our study have typing rules to address type checking issues as discussed in §3.1. There are contract-checking tools (e.g., PyContracts Graham et al. (2010)) that can check simple non-ML contracts. So, we recommend writing a similar extension tool that supports `scipy`, `CSR matrix` type checking, etc. *Scikit-learn* users can avoid type errors using such an extension tool. Additionally, such extensions can enforce these contracts through static or dynamic analysis. To further verify our findings, we obtain two more measures: the average elapsed time between the post time of first response and the response that is accepted, and the percentage of time the first answer is accepted. We annotate this as *average elapsed time*, and the *first answer accepted* in Table 7. A low rate of the first answer marked as accepted and higher elapsed time would generally indicate a difficulty in contract comprehension. We found that this additional evidence also confirms our finding that F contracts are usually harder to comprehend. We notice a relatively lower rate for accepting the first answer and higher elapsed time between a successful resolution and an initial attempt for the findings presented.

### 3.4 Localizing Contract Violations to Pipeline Stages

This section groups APIs into categories depending on the ML pipeline stage (described in §2.4) to explore **RQ4**. Islam et al. (2019) report that even for a subclass of ML contract violations that leads to bugs, bug localization is very challenging. This motivated us to study the stage of the APIs. Our goal was to identify the pipeline stages where contracts are frequently violated. Figure 6 depicts the distribution of the locations where the ML API contract violation occurred.



**Finding 11 :** A significant chunk of the ML contract violation occurred during data preprocessing and model construction stages.

**Model Construction and Data Preprocessing.** We observe that 30.1% of contract violations occur during the model construction stage (across all *SO* posts for all libraries). As an example, the *SO* post 14 using *Keras* failed to use a `softmax` activation in the *final* layer but chooses the value

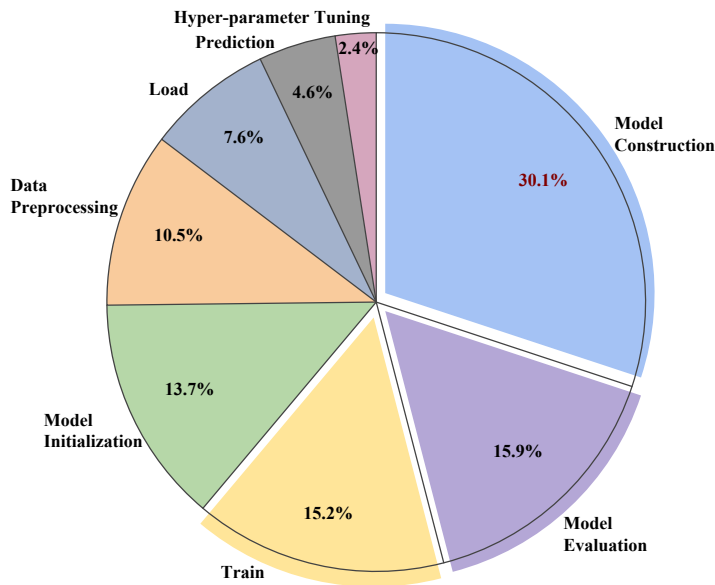
`categorical_crossentropy` as the *loss function* afterward. Here both the APIs involved, `Dense` and `Compile`, are from the model construction stage. In this case, the lack of contract checks results in the error propagating to the training and the prediction stages.

✓ Accepted Answer

**Your network will not work because of activation:** with `categorical_crossentropy` you need to have a `softmax` activation:

**Stack Overflow post 14:** Example post demonstrating contract violation in early ML pipeline stage

For *Scikit-learn* and *PyTorch*, we observe that 22.22% and 22.73% of errors occur respectively at the data pre-processing stage. Although it is one of the earliest stages in the ML pipeline, this trend is unique for these two libraries.



**Fig. 6:** Distribution of ML contract violation stages from *SO* posts

We observe that APIs from early pipeline stages are more susceptible to contract violations. This observation is crucial because ML pipelines often have inherent dependencies between pipeline stages. Violating contracts for APIs from early pipeline stages can lead to errors propagating to subsequent stages. We speculate from our data that further investigation in this area is needed. For instance, a possible future direction could be designing a verification system (as in prior work Sankaran et al. (2017); Dvijotham et al. (2019)) with ML contract knowledge. Contracts could explain cases where a bug in the ML system is caused by an API that has a location early in the pipeline com-

pared to where the error is registered. Catching the errors early in an ML system can enable better performance and help reduce costs.



**Finding 12 :** Training and model evaluation related APIs are the succeeding common locations that lead to contract violations across ML libraries.

**Train and Model Evaluation.** Training is one of the stages across all libraries that are prone to contract violation. 15.2% of the contract violations occur in APIs designed to train models. One of the primary reasons behind this is that current ML API documentation is insufficient on the topic of effects of optional parameters on the model’s accuracy rate. Contracts can document the appropriate relationship in regard to accuracy for such optional parameters in ML APIs found in our study. As an instance, one *SO* post<sup>8</sup> author talks about using the *Scikit-learn* API `linear_model.SGDClassifier.partial_fit()` due to dealing with the large size of training data. However, the ML API user was unaware that the required contract is to shuffle the data provided as the arguments for this API. User’s unawareness here can be considered towards insufficient documentation. Additionally, we observe a rate of 15.9% in terms of model evaluation stage related contract violations. Training and model evaluation stages are significantly important since, together, they can explain the trustworthiness of the model.



**Finding 13 :** Model initialization as contract violation stage is mostly prevalent in DNN libraries.

**Model Initialization.** Model initialization is the stage where DNN APIs are susceptible to violating contracts. The contract violations at this stage generally show a correlation towards the crash and bad performance effects. An example of model initialization stage discussed in one *SO* post<sup>9</sup>, the argument for `keras.backend.set_session` API should be a *TensorFlow session*. We group this example under the model initialization stage since the API in question sets up the environment. Automated tools (e.g., Auto-Net Mendoza et al. (2019)) that can build DNNs without human interventions can make use of API contracts from this stage to perform better and avoid crashes.

### 3.5 Threats to Validity

**Internal threats.** The first internal threat to the validity of our results is the classification scheme we used to identify ML contracts. To alleviate this threat, we have prepared the classification on top of well-established contract categories Pradel and Gross (2009); Khairunnesa et al. (2017); Leavens et al.

<sup>8</sup> <https://stackoverflow.com/questions/24617356/>

<sup>9</sup> <https://stackoverflow.com/questions/47167630/>



(2006). We have followed an open coding scheme only to add categories novel and ML-specific. The group effort to create the categories helped to make consistent choices.

To avoid the internal threat of bias in labeling attempts after training, the labelers performed an independent study, and the Kappa coefficient was used to measure inter-rater agreement. A moderator was present during the reconciliation of disagreements between raters.

**External threats.** The first external threat to validity is the reliability of the dataset we have used to conduct the empirical study. There are two sources of this threat: data source and data quality. For the data source, we have collected our data from a popular Q&A forum for software developers, *StackOverflow* (*SO*). *SO*, as a forum, maintains a strict moderation policy, promotes a peer-reviewing mechanism, and incorporates a reward system for encouraging quality answers from the developers<sup>10</sup>. Moreover, the latest software developer survey on the usage of *SO* forum reveals that its users come from all walks of managerial hierarchy, countries, experiences, age groups, expertise, races, etc<sup>11</sup>. As such, a huge user base, an abundance of topics, and a way to benchmark the quality of the contents make *SO* a frequent source in many SE studies Zhang et al. (2018a); Cai et al. (2019); Aghajani et al. (2019); Beyer and Pinzger (2014); Barua et al. (2012); Rosen and Shihab (2015); Cummaudo et al. (2020). Therefore, *SO* represents real-world ML developers (ML API users) and their concerns well and makes an ideal candidate for our study.

Next, to ensure good quality posts, we have gathered *SO* posts that have a high enough score Islam et al. (2019) in terms of questions and includes an accepted answer.

We have collected *SO* posts from four top ML libraries; however, the number of posts that we collected varies by the library. To measure the impact of this imbalance in the dataset, we have performed a two-tail test (inequality test) on the contract types for each library. Here, based on the *t-Stat*, and *t-Critical-two-tail* values,  $0.178 < 3.182$ , the observed difference between the sample means is small enough to say that the average number of contracts obtained from the four different ML libraries do not differ significantly. This result indicates that even though the dataset seems unbalanced in terms of the posts' frequency, the contract distribution is not unbalanced to a statistically significant extent. Additionally, prior works have Treude and Robillard (2016); Ellmann (2017) recognized *SO* as an important source to extract documentation for other domains. Multiple factors have enhanced the collected *SO* post used in this study compared to these prior related works. For example, the *SO* posts collected were from the years 2008 and 2021 and thus contained more recent posts than those used for the paper's submission. We also added some additional filtering criteria that suited the paper's main goal, e.g., the accepted answer post has a score higher than five and was required

<sup>10</sup> <https://stackoverflow.com/help/reputation>

<sup>11</sup> <https://survey.stackoverflow.co/2022/>

to contain code snippets or description that potentially describes a contract. We furthermore note that we closely followed the guidelines from prior works to conduct our study; however, there must have been some common SO posts that these previous works have studied.

Next, the nature of the methodology requires extensive manual work; thus, the number of libraries we could study is another closely related external threat. To lessen this threat, we have studied the highly-discussed four ML libraries based on SO trends since 2008. We have also observed that the number of curated posts for other ML libraries are less significant compared to the libraries that we have studied. For example, the libraries *apache-spark-mllib* and *weka* have only fourteen and two posts, respectively, for which contracts are relevant.

Furthermore, the SO posts are mostly about contract violations, and the answer posts talk about the needed contracts posing another external threat. If certain contracts (or violations) are not present in the dataset, our study will not find them. In essence, this is an out-of-the-vocabulary problem that is common in data mining techniques. Another possible external threat source is the need for validating findings with surveys and software developer interviews. While additional validations could raise confidence in the results, it is mitigated by the strict filtering criteria we use. We only look at the answers where at least five more users agree with the answers than those that disagree (which have been used in the past as a measure of the reliability of the answer Islam et al. (2019)). Moreover, we ensure that the only accepted answers by the questioner are studied. Thus, our filtering ensures that the derived contracts reflect a consensus among the questioner, the responder (via *acceptance tag*), and at least five users (minimum answer score of 5).

Finally, we must also consider ML API users expertise in our dataset as a threat to external validity. We have used a reliability score to mitigate this threat. Instead of using the general expertise of a software developer, the reliability score measures expertise on ML libraries. In short, the expertise of an user counts if they have earned the reputation from answering ML library related questions.

## 4 Related Work

No previous empirical studies have investigated the requirements for ML API contracts, but some prior work studied related issues.

**Studies of Bugs in ML Programs.** Zhang et al. (2018b) and Islam et al. (2019) have studied bugs for different DNN libraries using two sources: *Github* and *SO*. They have studied frequent bugs found in DNN libraries, root causes, and effects of these bugs. Humatova et al. (2020) presented a broad taxonomy of faults that occur in ML systems. To that end, they have surveyed ML developers in addition to studying code from *Github* and *SO*. Their taxonomy contains a category *API* that broadly categorizes usage faults of ML APIs. However, this category is too general to apprehend different types of API contracts. A recent work by Islam et al. (2020) studied the challenges

DNN developers face as they debug and subsequently examined the adopted bug fix patterns. Thung et al. (2012) performed an empirical study on general ML libraries. In addition to this, the study by Jia et al. (2020) examines the bugs found in *TensorFlow* programs. However, all of these prior works only present a classification for bugs; they do not identify the types of contracts that would prevent such bugs. In contrast, we focus on the contracts that the APIs from these libraries require and present a classification to identify different types of contracts. Contracts differ from bug patterns in that contracts do not just document incorrectness; they capture conditions needed to ensure correct behavior. Contracts can also be used to assign blame: if the client violates the contract for an API, then the client is to blame for incorrectness/bug in the software. On the other hand, if the client satisfies its part of an API contract, but the API does not satisfy its part, then the API’s implementation itself is buggy.

**Classification of Contracts.** The notion of contracts for APIs is well-established. Essentially two kinds of contracts, behavioral and temporal, are most often discussed in the literature Pradel and Gross (2009); Păsăreanu and Rungta (2010); Nguyen et al. (2014); Khairunnesa et al. (2017); Gruska et al. (2010); Nguyen et al. (2009); Wasylkowski et al. (2007). These two classes are behavioral and temporal contracts. In our work, we build upon these classes of contracts and explored their application to ML library APIs. Building on an existing classification scheme helped us not to reinvent known ideas Glaser (1978) related to API contracts. We also highlighted the new categories of specifications that are different than the non-ML APIs.

A recent study Leavens et al. (2022) points out the lessons we have learned in the course of the JML projects. It helps to design specification languages and tools for object-oriented languages such as Java and other languages. However, this work does not provide insight into the classes of contracts that Machine learning APIs require and their similarity and dissimilarity to traditional contracts that our work focuses on. Another research Pandita et al. (2012) proposes a technique to infer formal contracts from the natural language text of API documents. Such methodology will not suffice for ML APIs as we illustrate that most ML software exhibits crashes, and includes bad performance and incorrect functionality not obtained in the API documentation. Hence, we studied *SO* posts and characterized the types of ML contracts. Recently, Xie et al. (2022) proposed a technique to extract DL-specific input constraints from API documentation and to test APIs guided by such input constraints. However, our study pointed out that there are other kinds of contracts specific to ML, such as temporal contracts found in model architecture or other inter and intra-argument contracts, which could still be investigated further in the ML domain.

## 5 Conclusion and Future Work

ML has been applied in many software systems, including critical systems. However, like non-ML software, ML software can also be buggy. The presence of bugs gives rise to the problem of improving the reliability of software that uses ML libraries. ML software can suffer degradation of reliability in a statistical sense that may not cause obvious failures, thus detecting improper use of ML APIs can help improve its reliability. This motivated us to perform a comprehensive study to understand the types of contracts needed for ML APIs. Our study provides a taxonomy for ML API contracts and for violation location of these contracts. In this study, the question posts provided us with the ML API contract violations and the accepted answer posts contained the contracts. The frequent contract violations by the ML API users indicates the type of contracts that require immediate support. We have extracted 413 informal ML API contracts. End-users, including people teaching the application of ML libraries, can directly use the informal contracts from our study, as informal API documentation. The *SO* questions indicate a need for such contracts. Additionally, language designers can use these informal contracts as examples. The extracted contracts are labeled with the taxonomy presented in this paper. To help ML API users, libraries can be released with contracts enforced leveraging this taxonomy.

Our study has presented several key insights. First, many required contracts for ML libraries are not different than traditional contracts. However, ML API users struggle to maintain these contracts due to lack of domain knowledge, incomplete or ambiguous documentation, etc. Second, there are distinct ML-specific contracts, e.g., ML type checking. Additionally, ML APIs demonstrate a coupling between behavioral contracts and temporal contracts. Moreover, the uniqueness of these contracts allow the client to choose either temporal ordering or a state change. Third, ML API users struggle with maintaining temporal method orders (especially “eventually” constraints) for ML APIs. Fourth, ML API users often fail to satisfy input-related contracts of ML APIs, making input violations the most frequent root cause of contract violations in ML APIs. Fifth, when the ML contract violations lead to system failures, the error messages are often inadequate. Finally, a high percentage of contract violation occurs at early ML pipeline stages. In essence, the contract violation in an ML API that is used in early pipeline stages may delegate the effect in subsequent pipelines. The ML APIs from model construction, data preprocessing, etc. can benefit more from supporting contract checking compared to ML APIs that are used in later pipeline stages.

From this study, we envision several future directions. The classification described in our study could be used to design ML contract specification and verification tools. Such tools could help avoid or detect API-related bugs in ML programs or certify that an ML program is correct. An understanding of contract violations’ root causes and effects described in this paper could enable better debugging mechanisms and help detect contract violations. Comprehending the difficulty of resolving certain ML contract violations can help in

designing a recommendation system for ML API users. For instance, a recommendation system to automatically assign difficult contract violation related questions to expert users can be designed. Finally, understanding why ML API users make contract violations can help the designers of ML libraries to develop APIs that are easier to use and less prone to error.

**Acknowledgements** This material is based upon work supported by the National Science Foundation under Grant CCF-15-18897, CNS-15-13263, CCF-17-23215, CCF-17-23432, CCF-19-34884, CNS-17-23198, CCF-22-23812. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

### Data Availability Statement

The dataset (labeled *SO* posts, queries, source codes, etc.) generated during the study are available in the *figshare* repository, <https://figshare.com/s/c288c02598a417a434df>.

### Conflict of interest

The authors declare conflict of interest with the people affiliated with Iowa State University, University of Central Florida, and Bradley University.

### References

- Aghajani E, Nagy C, Vega-Márquez OL, Linares-Vásquez M, Moreno L, Bavota G, Lanza M (2019) Software documentation issues unveiled. 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE) pp 1199–1210
- Barua A, Thomas SW, Hassan AE (2012) What are developers talking about? an analysis of topics and trends in Stack Overflow. *Empirical Software Engineering* 19:619–654
- Beyer S, Pinzger M (2014) A manual categorization of android app development issues on Stack Overflow. 2014 IEEE International Conference on Software Maintenance and Evolution pp 531–535
- Cai L, Wang H, Xu B, Huang Q, Xia X, Lo D, Xing Z (2019) AnswerBot: An answer summary generation tool based on Stack Overflow. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2019, p 1134–1138, DOI 10.1145/3338906.3341186, URL <https://doi.org/10.1145/3338906.3341186>
- Chatterjee P, Kong M, Pollock L (2020) Finding help with programming errors: An exploratory study of novice software engineers’ focus in Stack

- Overflow posts. *Journal of Systems and Software* 159:110454, DOI <https://doi.org/10.1016/j.jss.2019.110454>, URL <http://www.sciencedirect.com/science/article/pii/S0164121219302286>
- Corbin J, Strauss A (1990) Grounded theory research: Procedures, canons and evaluative criteria. *Zeitschrift für Soziologie* 19(6):418 – 427, DOI <https://doi.org/10.1515/zfsoz-1990-0602>, URL <https://www.degruyter.com/view/journals/zfsoz/19/6/article-p418.xml>
- Corbin J, Strauss A (2008) *Basics of qualitative research* (3rd ed.): Techniques and procedures for developing grounded theory
- Cousot P, Cousot R, Fahndrich M, Logozzo F (2013) Automatic inference of necessary preconditions. In: in *Proceedings of the 14th Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'13)*, Springer Verlag, URL <https://www.microsoft.com/en-us/research/publication/automatic-inference-of-necessary-preconditions/>
- Cummaudo A, Vasa R, Barnett SA, Grundy J, Abdelrazek M (2020) Interpreting cloud computer vision pain-points: A mining study of Stack Overflow. *ArXiv abs/2001.10130*
- Dvijotham KD, Stanforth R, Goyal S, Qin C, De S, Kohli P (2019) Efficient neural network verification with exactness characterization. In: *Proc. Uncertainty in Artificial Intelligence, UAI*, p 164
- Ellmann M (2017) On the similarity of software development documentation. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2017*, p 1030–1033, DOI 10.1145/3106237.3119875, URL <https://doi.org/10.1145/3106237.3119875>
- Endres DM, Schindelin JE (2003) A new metric for probability distributions. *IEEE Transactions on Information theory* 49(7):1858–1860
- Glaser B (1978) Theoretical sensitivity. *Advances in the Methodology of Grounded Theory* URL <https://ci.nii.ac.jp/naid/10028142446/en/>
- Graham B, Furr W, Kuczmarski K, Biskup B, Palay A (2010) *Pycontracts*. <https://andreacensi.github.io/contracts/>, URL <https://andreacensi.github.io/contracts/>
- Gruska N, Wasykowski A, Zeller A (2010) Learning from 6,000 projects: Lightweight cross-project anomaly detection. In: *Proceedings of the 19th International Symposium on Software Testing and Analysis, Association for Computing Machinery, New York, NY, USA, ISSTA '10*, p 119–130, DOI 10.1145/1831708.1831723, URL <https://doi.org/10.1145/1831708.1831723>
- Guo, Yufeng (2017) 7 Steps of ML. URL <https://towardsdatascience.com/the-7-steps-of-machine-learning-2877d7e5548e>, retrieved Aug 2020
- Hoare CAR (1969) An axiomatic basis for computer programming. *Commun ACM* 12(10):576–580, DOI 10.1145/363235.363259, URL <https://doi.org/10.1145/363235.363259>
- Höst M, Wohlin C, Thelin T (2005) Experimental context classification: Incentives and experience of subjects. In: *Proceedings of the 27th International Conference on Software Engineering, Association for Computing Machinery,*

- New York, NY, USA, ICSE '05, p 470–478, DOI 10.1145/1062455.1062539, URL <https://doi.org/10.1145/1062455.1062539>
- Humbatova N, Jahangirova G, Bavota G, Riccio V, Stocco A, Tonella P (2020) Taxonomy of real faults in deep learning systems. In: ICSE'20: The 42nd International Conference on Software Engineering
- Islam MJ, Nguyen G, Pan R, Rajan H (2019) A comprehensive study on deep learning bug characteristics. In: ESEC/FSE'19: The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), ESEC/FSE 2019
- Islam MJ, Pan R, Nguyen G, Rajan H (2020) Repairing deep neural networks: Fix patterns and challenges. In: ICSE'20: The 42nd International Conference on Software Engineering
- Jia L, Zhong H, Wang X, Huang L, Lu X (2020) An empirical study on bugs inside tensorflows. In: Proc. DASFAA, p to appear
- Jothimurugan K, Alur R, Bastani O (2019) A composable specification language for reinforcement learning tasks. In: Advances in Neural Information Processing Systems, pp 13021–13030
- Khairunnesa SS, Nguyen HA, Nguyen TN, Rajan H (2017) Exploiting implicit beliefs to resolve sparse usage problem in usage-based specification mining. In: OOPSLA'17: The ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA'17
- Le TDB, Lo D (2018) Deep specification mining. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, Association for Computing Machinery, New York, NY, USA, ISSTA 2018, p 106–117, DOI 10.1145/3213846.3213876, URL <https://doi.org/10.1145/3213846.3213876>
- Le Goues C, Nguyen T, Forrest S, Weimer W (2012) GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering* 38(1):54–72
- Leavens GT, Baker AL, Ruby C (2006) Preliminary design of JML: A behavioral interface specification language for Java. *SIGSOFT Softw Eng Notes* 31(3):1–38, DOI 10.1145/1127878.1127884, URL <https://doi.org/10.1145/1127878.1127884>
- Leavens GT, Cok DR, Nilizadeh A (2022) Further lessons from the jml project. In: *The Logic of Software. A Tasting Menu of Formal Methods*, Springer, pp 313–349
- Lehtosalo J (2012) mypy. "<http://mypy-lang.org/index.html>", URL "<http://mypy-lang.org/index.html>", retrieved Aug 2020
- Lemieux C (2015) Mining temporal properties of data invariants. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, IEEE, vol 2, pp 751–753
- Lemieux C, Park D, Beschastnikh I (2015) General LTL specification mining (T). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, pp 81–92
- Li Y, Wang S, Nguyen TN (2020) DLFix: Context-based code transformation learning for automated program repair. In: ICSE'20: The 42nd International

- Conference on Software Engineering
- Long F, Rinard M (2015) Staged program repair with condition synthesis. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2015, p 166–178, DOI 10.1145/2786805.2786811, URL <https://doi.org/10.1145/2786805.2786811>
- Manna Z, Pnueli A (1992) The Temporal Logic of Reactive and Concurrent Systems. SV, NY
- Mechtaev S, Yi J, Roychoudhury A (2016) Angelix: Scalable multiline program patch synthesis via symbolic analysis. In: Proceedings of the 38th International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA, ICSE '16, p 691–701, DOI 10.1145/2884781.2884807, URL <https://doi.org/10.1145/2884781.2884807>
- Mendoza H, Klein A, Feurer M, Springenberg JT, Urban M, Burkart M, Dippel M, Lindauer M, Hutter F (2019) Towards automatically-tuned deep neural networks. In: Automated Machine Learning, Springer, pp 135–149
- Meyer B (1988) Object-oriented Software Construction. Prentice Hall, NY
- Meyer B (1992) Applying “design by contract”. Computer 25(10):40–51, DOI 10.1109/2.161279, URL <https://doi.org/10.1109/2.161279>
- Murali V, Chaudhuri S, Jermaine C (2017) Bayesian specification learning for finding API usage errors. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2017, p 151–162, DOI 10.1145/3106237.3106284, URL <https://doi.org/10.1145/3106237.3106284>
- Murphy KP (2012) Machine learning: a probabilistic perspective. MIT press
- Nasehi SM, Sillito J, Maurer F, Burns C (2012) What makes a good code example?: A study of programming q amp;a in stackoverflow. In: 2012 28th IEEE International Conference on Software Maintenance (ICSM), pp 25–34, DOI 10.1109/ICSM.2012.6405249
- Nguyen HA, Dyer R, Nguyen TN, Rajan H (2014) Mining preconditions of APIs in large-scale code corpus. In: FSE'14: 22nd International Symposium on Foundations of Software Engineering, FSE'14
- Nguyen TT, Nguyen HA, Pham NH, Al-Kofahi JM, Nguyen TN (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 Symposium on The Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE '09, p 383–392, DOI 10.1145/1595696.1595767, URL <https://doi.org/10.1145/1595696.1595767>
- Pandita R, Xiao X, Zhong H, Xie T, Oney S, Paradkar A (2012) Inferring method specifications from natural language api descriptions. In: 2012 34th International Conference on Software Engineering (ICSE), pp 815–825, DOI 10.1109/ICSE.2012.6227137
- Pei Y, Furia CA, Nordio M, Wei Y, Meyer B, Zeller A (2014) Automated fixing of programs with contracts. IEEE Transactions on Software Engineering 40(5):427–449, DOI 10.1109/TSE.2014.2312918



- Pradel M, Gross TR (2009) Automatic generation of object usage specifications from large method traces. In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society, USA, ASE '09, p 371–382, DOI 10.1109/ASE.2009.60, URL <https://doi.org/10.1109/ASE.2009.60>
- Păsăreanu CS, Rungta N (2010) Symbolic PathFinder: Symbolic execution of Java bytecode. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, Association for Computing Machinery, New York, NY, USA, ASE '10, p 179–180, DOI 10.1145/1858996.1859035, URL <https://doi.org/10.1145/1858996.1859035>
- Reger G, Barringer H, Rydeheard D (2013) A pattern-based approach to parametric specification mining. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, pp 658–663
- Rosen C, Shihab E (2015) What are mobile developers asking about? a large scale study using Stack Overflow. *Empirical Software Engineering* 21:1192–1223
- Sankaran A, Aralikkatte R, Mani S, Khare S, Panwar N, Gantayat N (2017) DARVIZ: deep abstract representation, visualization, and verification of deep learning models. In: 2017 IEEE/ACM 39th International Conference on Software Engineering: New Ideas and Emerging Technologies Results Track (ICSE-NIER), IEEE, pp 47–50
- Sarker S, Lau F, Sahay S (2000) Building an inductive theory of collaboration in virtual teams: an adapted grounded theory approach. In: Proceedings of the 33rd Annual Hawaii International Conference on System Sciences, pp 10 pp. vol.2–
- Seshia SA, Desai A, Dreossi T, Fremont DJ, Ghosh S, Kim E, Shivakumar S, Vazquez-Chanlatte M, Yue X (2018) Formal specification for deep neural networks. In: International Symposium on Automated Technology for Verification and Analysis, Springer, pp 20–34
- Sim J, Wright CC (2005) The Kappa Statistic in Reliability Studies: Use, Interpretation, and Sample Size Requirements. *Physical Therapy* 85(3):257–268, DOI 10.1093/ptj/85.3.257, URL <https://doi.org/10.1093/ptj/85.3.257>, <https://academic.oup.com/ptj/article-pdf/85/3/257/31670498/ptj0257.pdf>
- StackOverflow Reputation (2023) StackOverflow Reputation and Moderation. URL <https://stackoverflow.com/help/reputation>, retrieved Jan 2023
- StackOverflow Survey (2017) Survey. URL <https://survey.stackoverflow.co/2022/>, retrieved Jan 2023
- Sun X, Zhou T, Li G, Hu J, Yang H, Li B (2017) An empirical study on real bugs for machine learning programs. In: 2017 24th Asia-Pacific Software Engineering Conference (APSEC), pp 348–357, DOI 10.1109/APSEC.2017.41
- Thung F, Wang S, Lo D, Jiang L (2012) An empirical study of bugs in machine learning systems. In: Proceedings of the 2012 IEEE 23rd International Symposium on Software Reliability Engineering, IEEE Computer Society, USA, ISSRE '12, p 271–280, DOI 10.1109/ISSRE.2012.22, URL

- <https://doi.org/10.1109/ISSRE.2012.22>
- Treude C, Robillard MP (2016) Augmenting API documentation with insights from Stack Overflow. In: Proceedings of the 38th International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA, ICSE '16, p 392–403, DOI 10.1145/2884781.2884800, URL <https://doi.org/10.1145/2884781.2884800>
- Viera AJ, Garrett JM, et al. (2005) Understanding interobserver agreement: the kappa statistic. *Fam med* 37(5):360–363
- Wang S, Chollak D, Movshovitz-Attias D, Tan L (2016) Bugram: Bug detection with n-Gram language models. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, Association for Computing Machinery, New York, NY, USA, ASE 2016, p 708–719, DOI 10.1145/2970276.2970341, URL <https://doi.org/10.1145/2970276.2970341>
- Wasylkowski A, Zeller A, Lindig C (2007) Detecting object usage anomalies. In: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC-FSE '07, p 35–44, DOI 10.1145/1287624.1287632, URL <https://doi.org/10.1145/1287624.1287632>
- Xie D, Li Y, Kim M, Pham HV, Tan L, Zhang X, Godfrey MW (2022) Doc-ter: Documentation-guided fuzzing for testing deep learning api functions. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Association for Computing Machinery, New York, NY, USA, ISSTA 2022, p 176–188, DOI 10.1145/3533767.3534220, URL <https://doi.org/10.1145/3533767.3534220>
- Zhang T, Upadhyaya G, Reinhardt A, Rajan H, Kim M (2018a) Are code examples on an online Q&A forum reliable? a study of API misuse on Stack Overflow. In: Proceedings of the 40th International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA, ICSE '18, p 886–896, DOI 10.1145/3180155.3180260, URL <https://doi.org/10.1145/3180155.3180260>
- Zhang T, Gao C, Ma L, Lyu M, Kim M (2019) An empirical study of common challenges in developing deep learning applications. In: 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE), pp 104–115, DOI 10.1109/ISSRE.2019.00020
- Zhang T, Gao C, Ma L, Lyu M, Kim M (2019) An empirical study of common challenges in developing deep learning applications. In: 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE), IEEE, pp 104–115
- Zhang Y, Chen Y, Cheung SC, Xiong Y, Zhang L (2018b) An empirical study on TensorFlow program bugs. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, Association for Computing Machinery, New York, NY, USA, ISSTA 2018, p 129–140, DOI 10.1145/3213846.3213866, URL <https://doi.org/10.1145/3213846.3213866>

---

Zhong H, Meng N, Li Z, Jia L (2020) An empirical study on API parameter rules. In: ICSE'20: The 42nd International Conference on Software Engineering