Automated Testing of Software that Uses Machine Learning APIs

Chengcheng Wan
University of Chicago
cwan@uchicago.edu

Shicheng Liu
University of Chicago
shicheng2000@uchicago.edu

Sophie Xie
Whitney Young High School
sxie2@cps.edu

Yifan Liu
University of Chicago
liuyifan@uchicago.edu

Henry Hoffmann
University of Chicago
hankhoffmann@uchicago.edu

Michael Maire
University of Chicago
mmaire@uchicago.edu

Shan Lu
University of Chicago
shanlu@uchicago.edu

ABSTRACT

An increasing number of software applications incorporate machine learning (ML) solutions for cognitive tasks that statistically mimic human behaviors. To test such software, tremendous human effort is needed to design image/text/audio inputs that are relevant to the software, and to judge whether the software is processing these inputs as most human beings do. Even when misbehavior is exposed, it is often unclear whether the culprit is inside the cognitive ML API or the code using the API.

This paper presents Keeper, a new testing tool for software that uses cognitive ML APIs. Keeper designs a pseudo-inverse function for each ML API that reverses the corresponding cognitive task in an empirical way (e.g., an image search engine pseudo-reverses the image-classification API), and incorporates these pseudo-inverse functions into a symbolic execution engine to automatically generate relevant image/text/audio inputs and judge output correctness. Once misbehavior is exposed, Keeper attempts to change how ML APIs are used in software to alleviate the misbehavior. Our evaluation on a variety of open-source applications shows that Keeper greatly improves the branch coverage, while identifying many previously unknown bugs.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Computing methodologies** → **Machine learning**; • **Information systems** → RESTful web services.

KEYWORDS

software testing, machine learning, machine learning API

ACM Reference Format:
Chengcheng Wan, Shicheng Liu, Sophie Xie, Yifan Liu, Henry Hoffmann, Michael Maire, and Shan Lu. 2022. Automated Testing of Software that Uses Machine Learning APIs. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, P A, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510068

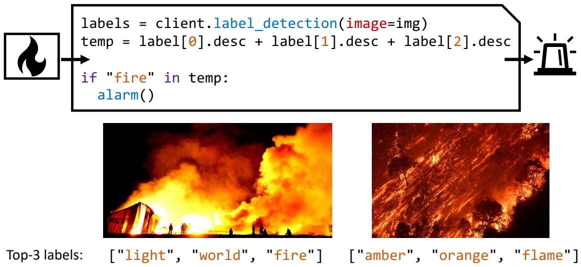
1 INTRODUCTION

1.1 Motivation

Machine learning (ML) offers powerful solutions to cognitive tasks, allowing computers to statistically mimic human behaviors in computer vision, language, and other domains. To facilitate easy use of these ML techniques, many cloud providers offer well-designed, well-trained, and easy-to-use cognitive ML APIs [1–5]. Indeed, many software applications in a variety of domains are incorporating ML APIs [6, 7]. Thus, effectively testing these applications—which this paper refers to as *ML software*—has become urgent.

To better understand this testing task, consider Phoenix [8], a fire-alarm application. As shown in the top half of Figure 1, Phoenix uses the Google `label_detection` API to perform image classification on an input photo, and then triggers an alarm if any of the top-3 classification labels returned by the API includes the keyword "fire".

This simple demo application turns out to be difficult to test. First, random inputs work poorly, as they rarely contain fire and hence cannot exercise the critical `alarm()` branch. Second, even with carefully collected image inputs, manual checking is likely needed to judge the execution correctness (i.e., whether an alarm should be triggered). Finally, even after a failed test run—e.g., the picture on the right of Figure 1 fails to trigger the alarm—it is difficult to know whether the failure is due to the statistical nature of `label_detection`, which has to be tolerated, or the application's incorrect use of the API, which has to be fixed. In fact, this case belongs to the latter: the right figure actually has a top-3 label "flame" returned by `label_detection`; not checking for the "flame" label, this application may miss fire alarms in many critical situations.



```
labels = client.label_detection(image=img)
temp = label[0].desc + label[1].desc + label[2].desc

if "fire" in temp:
  alarm()
```

Figure 1: An example of using ML Cloud APIs [8].

This example has demonstrated several open challenges in testing ML software.

1) Infinite, yet sparse input spaces. The spaces of images, texts, or audios —typical input forms of cognitive ML APIs—are infinitely large, yet *realistic* inputs that are *relevant* to the software-under-test

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510068

are spread sparsely throughout this space. For example, only a tiny portion of real-world images contain fire and are relevant to the fire alarm software.

Existing input generation techniques are ineffective here. Random input generators cannot produce realistic inputs through random-pixel images or random-character strings. Fuzzing techniques that apply perturbations (white noises [9], block replacement [10] , or mapping [11]) to seed inputs tend to produce inputs that are either unrealistic or similar with the seed. For example, no fuzzing can turn the left photo into the right photo in Figure 1. Symbolic execution techniques also do not work, as it is difficult to express the input realism as a solvable constraint. Furthermore, none of these techniques solves the relevance challenge. To tell which images are relevant for a fire alarm application requires both an understanding of the software structure (i.e., knowing that a branch predicate is about fire in the input) and the ability to perform the very cognitive task we need to test (i.e., judging whether a photo contains fire).

2) Output correctness relying on human judgement. Cognitive ML APIs are designed to statistically mimic human behaviors, e.g., identifying the objects in an image, interpreting the emotional sentiment in a sentence, etc. Consequently, to judge the correctness of ML software, ideally, we want to ask many people to process the same set of inputs and see if their decisions statistically match with the software outputs—a process that is inherently difficult to automate. For example, it is difficult to tell whether the fire alarm should be triggered or not without manual inspection (Figure 1).

In traditional testing, the execution correctness often can be checked automatically using the mathematical relationship between the inputs and the outputs or certain invariants expected to hold by the execution. These techniques are still useful for the non-cognitive parts of the ML software, but cannot help the cognitive parts. Previous work generated test oracles for domain-specific applications, like an image dilation software [12], a blood-vessel categorizer [13], an image region growth program [14], a biomedical text processor [15]. Their design each targets a particular cognitive task and cannot be applied for general ML software.

3) Probabilistic incorrectness that is difficult to diagnose. When ML software produces outputs that differ from most human beings' judgement, which we refer to as *an accuracy failure*, developers must attribute this failure to either the ML API or the surrounding software's use of the ML API. This attribution is difficult as ML APIs use statistical models to emulate cognitive tasks, and are expected to produce incorrect outputs from time to time. In other words, developers need to distinguish failures caused by the probabilistic nature of the ML API, which simply must be tolerated as part of using this specific ML API, from a misuse of the API, which represents a bug and must be fixed by the developer.

Again, this situation is different from that in traditional software testing, where a test failure like a crash indisputably points out something incorrect with the software that needs to be fixed.

Note that, much recent work studies how to test [9, 16–41] and fix [42–45] neural networks. However, they focus on improving the accuracy, fairness, and security of the neural network itself; e.g., making sure the network is robust against adversarial samples or does not contain certain biases, etc. They do *not* consider how the neural network is *used* in the context of an application and do *not* test how well the application using the neural network functions.

1.2 Contributions

This paper proposes Keeper, a testing tool designed for software that uses cognitive machine learning APIs (ML software).

To tackle the unique input space and output oracle challenges, Keeper designs a set of pseudo-inverse functions for cognitive ML APIs[1]. For an API f that maps inputs from domain $\mathbb{I}$ to outputs in domain $\mathbb{O}$, its pseudo-inverse function f' reverses this mapping at the semantic level. We make sure that the mapping by f' has been confirmed by many people to have high accuracy. For example, the Bing image search engine is a pseudo-inverse function of Google's image classification API.

Keeper then integrates the pseudo-inverse functions with symbolic execution to reach the sparse program-relevant input space. Specifically, Keeper first uses symbolic execution to figure out what values an ML-API output can take to fulfill branch coverage (e.g., `"fire"` == `labels[0].desc` in Figure 1). Keeper then automatically generates realistic inputs that are expected to produce the desired ML-API outputs, leveraging pseudo-inverse functions. For example, the two images shown in Figure 1 are among the images returned by a Bing image search with the keyword "fire".

Keeper also makes pseudo-inverse functions a proxy of human judgement and automatically judges the correctness of software outputs that are related to cognitive tasks. Since our pseudo-inverse functions are *not* analytically inverting ML APIs (i.e., $f'(f(i)) \neq i$ is possible), a test input generated by Keeper may not cover the targeted software branch, like the right image in Figure 1 failing to cover the `alarm` branch. At the same time, since these pseudo-inverse functions have been approved by many human beings, Keeper reports an *accuracy failure* when over a threshold portion of inputs fail to cover a particular target branch.

Of course, Keeper also monitors generic failure symptoms like crashes during test runs, and helps expose bugs in code regions that require specific ML inputs to exercise.

Finally, to help developers understand the root cause of an accuracy failure, Keeper explores alternative ways of using ML APIs and informs the developers of any code changes that can alleviate the accuracy failure. For the example in Figure 1, Keeper would inform developers that comparing the returned labels with not only "fire" but also "flame" would make the software behavior more consistent with common human judgement.

Putting these all together, we have implemented Keeper that can be used either through a command-line script or a plug-in inside the VScode IDE [46]. Given a software application, Keeper first highlights all the functions that directly or indirectly call ML APIs. For any function that developers want to test, Keeper automatically generates many test cases to thoroughly test every branch in the specified function and its callees. Keeper analyzes the test runs and reports any failures, as well as potential patches for accuracy failures, to developers.

We evaluate Keeper on the latest version of 63 open-source Python applications that cover different problem domains and ML APIs. Due to the relatively young age of ML APIs, these 63 applications are mostly research projects, hackathon products, and demo programs. Keeper achieves 91% branch coverage on average for

[1] The current implementation of Keeper supports Google Cloud AI APIs and can be easily extended to support similar APIs from other service providers.

these applications. In total, Keeper covers 21–38% more branches than alternative techniques that directly use machine learning training data set or random fuzzing. Keeper exposes 35 unique accuracy and crash failures from 25 out of these 63 applications.

2 BACKGROUND AND DEFINITIONS

This section provides a brief overview of ML APIs, their inputs and outputs, and how they are typically used in software.

ML Cloud APIs. ML APIs offered by different service providers all cover three main categories of machine learning tasks: vision tasks, language tasks, and speech tasks. Keeper handles all the commonly used APIs in these three families, as shown in Table 1. Keeper currently does not handle Video Intelligence APIs (from the vision family), Translation APIs (from the language family), and Speech Synthesis APIs (from the speech family), as they are used much less frequently in open-source applications [7].

In addition to image/text/audio inputs, some ML APIs also take in configuration parameters. For example, analyze_sentiment takes in not only a text string, but also configurations like language, encoding, and input type, as shown in Figure 2. These configurations are set to constant values, mostly the default values offered by Google, in all of the ML software we have checked. Therefore, in this paper, Keeper focuses on generating image/text/audio inputs.

```
document = {"content": text_content, "type_": Type.
    PLAIN_TEXT, "language": "en"}
response = client.analyze_sentiment(request= {'document':
    document, 'encoding_type': EncodingType.UTF8})
```

Figure 2: An example of Google Cloud API with text input.

The output of a ML API may include multiple records, like multiple classification results, multiple objects detected, and so on. Each record typically contains a key result field often of a string or an enum type, like the classification label of an image, the emotion of a face, and so on, and a confidence score field, which indicates how likely this result is correct. Unless otherwise specified, the remaining paper refers to these key result fields as *ML API output*, as summarized in Table 1. Note that, some of these APIs do output other auxiliary information. For example the face detection API also outputs the bounding box of each face detected in the input image. These auxiliary result fields may be used to make control flow decisions, although such usage has not been observed in any of the 360 applications collected by a previous ML API study [7].

ML software. Sometimes, ML APIs are only loosely connected with the remaining part of the software, with their output directly printed out without further use in the software. Testing this type of software simply needs to test ML APIs and the remaining part of the software separately and hence is not the target of Keeper.

In some other cases, ML APIs are more closely connected, with their results used to impact the control flow of the software execution. These cases present new challenges to software testing as discussed in Section 1 and hence is the focus of this paper.

3 TEST INPUT GENERATION

Keeper is a testing tool for software whose control flow is influenced by ML APIs. As shown in Figure 3, Keeper includes two major components: test-input generation, which we present in this section, and test-output processing, which we present in Section 4.

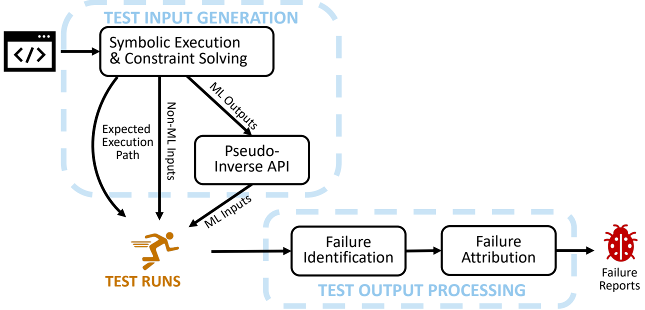


Figure 3: An overview of Keeper.

Keeper’s input generation is built upon an existing symbolic execution engine, DSE [47]. Given a function F to test[2] and all the function parameters represented as symbolic variables, a symbolic path constraint is generated for every branch; solving all the path constraints produces a test suite that offers full branch coverage.

In this section, we explain how Keeper handles cases when ML APIs are part of the path constraints and generates inputs for ML APIs, which are not handled by existing techniques.

A naive solution is to symbolically execute ML APIs’ implementation. Unfortunately, this is too expensive to carry out for state-of-the-art deep neural networks (DNN). Not to mention that the exact DNNs used by ML API providers are unknown. For example, a state of the art image classification network, EfficientNet-L2 [48], has 480 million parameters. It takes in a 224 x 224 pixel image and generates the output after about 50 billion floating point operations. Solving a path constraint that involves this network with more than 50,000 (224 x 224) symbolic variables would take days.

Keeper decomposes the problem of generating inputs for ML APIs into two parts: first, it identifies the ML-API outputs that are needed to satisfy path constraints using symbolic execution (Section 3.1); and then synthesizes the ML-API inputs that are expected to produce those outputs using carefully designed pseudo-inverse functions (Section 3.2). As we will see, this decomposition not only avoids the complexity of directly applying symbolic execution to DNNs, but also help judge the execution correctness (Section 4).

3.1 Identifying relevant ML outputs

To identify the desired ML-API outputs, Keeper makes its symbolic execution skip any statement that calls an ML API and instead marks API output that is used by following code as symbolic. This way, the output, instead of input, of ML APIs will be part of the path constraints, and by solving the constraints, Keeper obtains the API-output values that are needed to exercise corresponding branches.

The only tweak Keeper makes here is to have the symbolic execution engine sometimes generating one path constraint for each branch sub-condition, instead of the whole branch. Specifically, a common code pattern that we have observed is to decide the

[2] Users of Keeper can choose any function to test, including the main function.

ML Task		Main Output	Constraint Example	Pseudo-inverse Function
Vision	Image classification	image class	class=="fire" [8]	Search on internet, keyword: [image class]
	Object detection	object name	object=="tableware" [49]	Search on internet, keyword: [object name]
	Face detection	face emotion	emotion =="joy" [50]	Search on internet, keyword: [emotion] + "human face"
	Text detection	extracted text	text=="3923-6625" [51]	Print [extracted text] on an image
Language	Document classification	document class	class=="food" [52]	Search on internet, keyword: [document class]
	Sentiment detection	score, magnitude	score< 0 [53]	Select tweets from Sentiment140 dataset [54]
	Entity detection	entity name, type	type=="Person" [55]	Use text generation technique, seed: [name] or [type]
Speech	Speech recognition	transcript	text=="turn on the light" [56]	Use speech synthesize technique on [transcript]

Table 1: Different ML APIs handled by Keeper and their pseudo-inverse functions.

```
1 def smart_can(img):
2   labels = client.label_detection(image=img)
3   classes = [x.desc for x in labels]
4   for c in classes:
5     if  c == "food":
6       return "organic"
7     if c == "paper" or c == "aluminum":
8       return "recyclable"
9   return "non-recyclable
```

Figure 4: A smart can application, Heap-Sort-Cypher [57]

execution path based on whether or not an ML API outputs a label that belongs to a pre-defined set. For example, the smart-can application in Figure 4 executes the `recyclable` path when the output of `label_detection` contains a label that is either `paper` or `aluminum`. Since different labels often represent different types of real-world inputs, Keeper will generate one path constraint for every condition clause, instead of one for the whole branch. For example, for the line-7 branch in Figure 4, Keeper generates two constraints ("paper" ∈ `classes`) and ("aluminum" ∈ `classes`), which then prompts Keeper to generate two separate sets of images to satisfy these two constraints.

In our implementation, this is accomplished by enabling a corresponding feature of the underlying symbolic execution engine. For example, for a branch condition "A or B or C", four constraints will be formed representing (1) A is True, (2) B is True, (3) C is True, and (4) none of A, B, C is True. Solving these constraints leads to four inputs or input sets that satisfy these constraints separately.

3.2 Identifying ML API inputs

Given an ML API f and an output o, Keeper aims to automatically generate a set of inputs I so that $f(i), i \in I$ is *expected* to produce o according to common human judgement. For example, the two images in Figure 1 are expected to make `label_detection` output "fire" and the images in every column of Figure 5 are expected to make `label_detection` output the corresponding column-header.

To achieve this, Keeper designs a pseudo-inverse function f' for every API f, so that $f'(o)$ will produce the input set I for f. We want f' to have the following properties.

First, f' is not an analytical inversion of f. Ideally, f' should be built independently from f (e.g., not based on the same training data set), so that f' can help not only input generation but also failure identification in a way similar to N-version programming.

Second, f' should be a semantic inverse of f, reversing the cognitive task performed by f in a way that is consistent with most

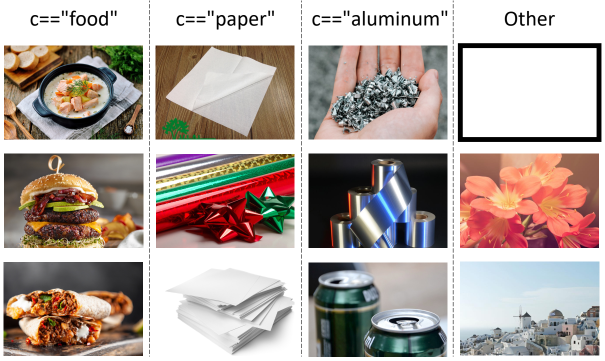


Figure 5: Keeper-generated test cases for Figure 4

human beings. This way, test inputs generated by Keeper can expect to cover most of the software branches, unless the ML API is unsuitable for the software or is used incorrectly.

Third, f' should produce more than one output for each input it takes in. This will allow Keeper to generate multiple inputs for f to exercise a corresponding branch, and get a statistically meaningful test result given the probabilistic nature of ML APIs.

With these goals in mind, we have designed three types of pseudo-inverse functions as summarized in Table 1.

3.2.1 Search-based pseudo inversion. For many vision and language APIs, search engines offer effective pseudo inversion: they take in a keyword and return a set of realistic images/texts that reflect the keyword. Search engines have several properties that serve Keeper's testing purposes. First, they offer great semantic inversion, as there are multiple search engines that have been used by hundreds of millions of users for many years with high satisfaction [58]. Their top search results typically match the common human judgement. Second, they are not an analytical inversion of ML APIs, and we will use non-Google engines to minimize potential correlations. Third, they accept a wide range of search words and produce many ranked results, which means a large number of high-quality test inputs for Keeper. Specifically, Keeper uses different engines and search keywords for different ML APIs:

Vision tasks. Image-classification and object-detection APIs return string labels that describe the image and the objects inside the image, respectively. For both APIs, Keeper uses the Bing [59] image search engine and uses the desired label description or object name as the search keyword. For example, the images in each column of Figure 5 were the top-3 search results returned by Bing using the keywords listed atop. The only exception is the last column: when there is no specific keyword requirement (like `c != food and c!= paper and c != aluminum`), Keeper uses a blank image and images generated by a random-image generator [60].

The face-detection API detects human faces in an image. Some ML software uses the returned emotion string associated with each face (e.g., "joy", "sorrow", etc.) to decide execution path. To generate corresponding images, Keeper uses "[emotion] human face" as a keyword to search the Bing image.

Language tasks. Document-classification APIs process a document and return categories based on the document content, like "pets", "health", "sports", and others. Keeper uses the desired category name as keyword and searches it at (1) knowledge graph websites, Wikipedia [61] and Britannica [62]; and (2) Bing web search engines. Keeper then uses the text extracted out from each returned web page as the ML API input.

3.2.2 *Synthesis-based pseudo inversion.*

The semantic inversion of some ML APIs does not match the functionality of search engines. Fortunately, we find ways to synthesize inputs for them.

The **text-detection** API extracts printed or handwritten text from an image. Unfortunately, image search engines tend to return images whose content reflects the search keyword, instead of images that contain the keyword as text within the image. Therefore, given a text string, Keeper prints it on a background image using the Python pillow library [63]. Keeper adopts both printed and hand-writing fonts; different font settings produce different test images. To decide the background image, Keeper checks whether the `text-detection` API shares its input image with another vision API. If so, the test images Keeper generated for the other API will be used as the background; otherwise, a blank image and some random images will be used. Figure 6 shows some of the test images that Keeper generates for application wanderStub [64], which has a branch checking if the input image contains "Total".

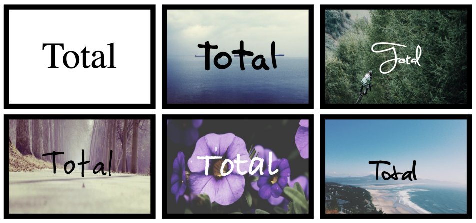


Figure 6: Test inputs generated for wanderStub [64].

The **entity-detection** API inspects the input sentence for known entities—there are in total 13 entities, such as ADDRESS, DATE, etc. Since the search engines usually return long documents, Keeper instead uses a popular language model GPT-2 [65] to synthesize any number of sentences that start with a pre-defined word/phrase that corresponds to the desired entity type.

The **speech-recognition** API transcribes the input audio clip and outputs the transcript. Keeper uses speech synthesis tools, particularly the pyttsx3 [66] Python library, to generate the desired audio clips based on a given transcript. Keeper generates multiple audio clips using different voice settings supported by this library.

3.2.3 *ML benchmarks for pseudo inversion.*

The **sentiment detection** API presents two challenges. First, although this API aims to identify the prevailing emotional opinion within the text, it does not directly output a categorical result. Instead, it returns two floating-point numbers, `score` and `magnitude`, for developers to derive emotion categories from. There is no perceivable way to generate text that can offer the exact `score` or `magnitude`. Second, even if we just hope to generate text that contains positive or negative emotion, no search engine or synthesizer can accomplish this.

Facing these challenges, Keeper resorts to the Sentiment140 dataset [54], which contains 1,600,000 tweets, manually labelled as positive, negative, and neutral. Keeper randomly samples the same number of positive, negative, and neutral tweets as test inputs for any sentiment-detection API called inside an ML software, with the expectation that these tweets will help cover different branches in the software that are designed for different emotions.

Note that, we treat ML benchmarks as the last resort for multiple reasons. First, the labels associated with data inside ML benchmarks either have few categories or have limited quality. For example, ImageNet [67] contains 1000 manually labeled image categories, which is too few compared with the 20,000 labels of Google Vision AI. On the contrary, OpenImage has 9 million images with 20,000 labels. However 89% of the labels are generated by DNNs, and 53% of the human-verified ones are incorrect [68]. Second, ML benchmarks are built with pre-processed real-world data. Such "clean" data has less variety, as they share similar size, resolution, and encoding format. Third, some benchmarks may be part of the training data set of Google ML APIs, which makes the test inputs biased towards the ones APIs can perform well on and hence less likely to reveal problems. Finally, Generative Adversarial Network synthesizes new data following the distribution of the training set [69]. It covers different domains, including generating images from text [70]. We do not use it, as this approach requires much training data and ends up generating non-real-world data that has similar distribution with the training set, whose limitations we discussed earlier.

3.3 Putting everything together

Overall, Keeper generates test inputs for any function F in the following steps. First, its symbolic execution (Section 3.1) generates a set of inputs $\mathbb{I}$ that offer full branch coverage unless some path constraints are un-satisfiable. If no branch in F or its callees depends on the output of an ML API, the input generation is done. Otherwise, if there is such an ML-dependent branch b, those inputs that are expected to cover b, denoted as $\mathbb{I}_b \subset \mathbb{I}$, contain fields that represent the desired outputs of ML APIs and require further processing.

Next, for each desired output o of an ML API f, Keeper applies f's pseudo-inverse function f' on o to generate a set of image/text/audio inputs for f (Section 3.2). If f's input is exactly an input of the function under test F (i.e., it is not derived from an input of F through pre-processing), the input generation is done. Keeper updates every input in $\mathbb{I}_b$ with the image/text/audio information. If there were k inputs in $\mathbb{I}_b$, Keeper now gets $k \times N_b$ inputs, with N_b being the number of image/text/audio inputs Keeper generated for the ML API f to exercise b. Developers can configure N_b, or the total number of test inputs to generate. Keeper will then compute N_b, so that every ML-dependent branch (sub-)condition gets exercised by about the same number of inputs.

If f's input is derived from an input of function F through pre-processing, Keeper runs symbolic execution on that pre-processing code to figure out the desired input of F and finishes the input generation. For example, if a function deletes the first character of a string parameter and feeds the resulting string to an ML API f, Keeper will add a character to the beginning of every input generated for f to get the string parameter of this function. The symbolic execution engine used by Keeper can handle pre-processing related to text (i.e. strings), but not those related to images or audio, such as image/audio clipping. Future work can extend Keeper with common image/audio transformation routines.

Finally, these test inputs generated by Keeper are ready to be executed. Particularly, in order for a software to consume a test image or audio file $file$ generated by Keeper, Keeper changes the file path embedded in the software to a path that points to $file$.

4 TEST OUTPUT PROCESSING

Once all the test inputs are generated and executed, Keeper works on failure identification and attribution.

4.1 Failure identification

Keeper looks for three types of failure symptoms: (1) low accuracy, (2) dead code, and (3) generic failures like crashes.

4.1.1 *Low-accuracy failures.*

When software incorporates cognitive ML APIs in its computation, judging the output's correctness becomes challenging: (1) by definition of cognitive tasks, this output needs to be checked with many people to see if it matches with common human judgement; (2) due to the probabilistic nature of ML APIs, an occasional mismatch is expected. Of course, frequent mismatches are un-acceptable and severely hurt user experience, like not triggering fire alarms when needed (Figure 1) or consistently categorizing garbage incorrectly (Figure 4).

To tackle the first challenge, Keeper uses pseudo-inverse functions as an approximation of common human judgement; to tackle the second challenge, Keeper considers the software to suffer from a low-accuracy failure, or an *accuracy failure* for short, only when over a threshold portion of inputs of a particular type have produced outputs that are inconsistent with common human judgement.

Specifically, for all the inputs $\mathbb{I}_b$ that are generated to cover a branch b, Keeper checks which of them exercise b at run time, denoted as $\mathbb{I}_b^{\text{succ}}$ and calculates the *recall* of b (i.e., $\frac{|\mathbb{I}_b^{\text{succ}}|}{|\mathbb{I}_b|}$). If the recall drops below a threshold α, 75% by default. Keeper reports an accuracy failure associated with b. The setting of α can be adjusted, but should not be 100%, as ML APIs are probabilistic and pseudo-inverse functions cannot guarantee to be correct all the time.

For the fire-alarm example in Figure 1, Keeper identifies an accuracy failure associated with the `"fire"` branch, as its recall is 41%; for the smart-can example in Figure 4, Keeper identifies an accuracy failure as the recall of the recyclable branch is only 13%.

For a branch b that depends on the output of a sentiment-detection API, Keeper identifies failures slightly differently as inputs are generated for sentiment-detection API differently as discussed in Section 3.2.3. During test runs, Keeper checks all the inputs that exercise b to see what portion of them are labeled as having positive emotion and what portion are labeled as negative. If both go above a threshold, indicating that branch b is not accurately differentiating inputs with different emotions, Keeper reports an accuracy failure.

```
labels = client.label_detection(image=img)
temp = label[0].desc + label[1].desc + label[2].desc
if "fire" in temp or "flame" in temp  or "ash" in temp:
  alarm()
```

Figure 7: A fixed version of Figure 1 suggested by Keeper.

Root causes of accuracy failures. Note that, these accuracy failures are **not** equivalent with low precision or low recall of the ML API itself. The latter is just one of the possible root causes of the former. Keeper intentionally does not calculate the precision or recall of any ML API, but instead focuses on the overall software.

One possible cause is that developers missed some related labels in a branch condition, which we refer to as an *incomplete label* problem. For example, the `label_detection` API does not return "fire" as a top-3 label for many top fire images returned by the Bing image search. This by itself is *not* considered a failure by Keeper. If the software uses the API properly, like raising a fire alarm upon not only a "fire" label but also a "flame" label and an "ash" label as shown in Figure 7, no accuracy failure would be reported, as the recall of the alarm-related branch is as high as 85% and the precision is 100% in our experiments.

Another possible cause is that developers used a non-existing label, which does not exist in the API's label set and can never be the output. This is not a surprise as the labels that can be output by Google Vision API are too many (19,985) for developers to memorize. For example, an application compares the `label_detection` output with "clothes" and "pants" [57], which are non-existing labels. Instead, "clothing" and "trousers" are valid labels.

4.1.2 *Dead-code failures.*

These occur when a branch is not covered after all the testing runs. They happen under two scenarios.

One scenario is that Keeper generates a set of test inputs $\mathbb{I}_b$ expected to cover a branch b, and yet b is not exercised by any input in $\mathbb{I}_b$. Such an extreme case of low branch recall (i.e., 0) is often caused by the branch comparing a ML API output with a non-existing label. If this comparison is one of multiple branch sub-conditions, an accuracy failure would likely occurr (i.e., a low but non-zero recall); if it is the only condition clause, a deadcode failure occurs. For example, a smart photo application FESMKMITL [71] checks the output of `label_detection` against the string "face". Unfortunately, among the 20,000 category labels that could be output by this API, none of them is "face". Instead, "human face" is one of the valid labels for this API, which the developers should have used.

The other scenario is that Keeper fails to generate any inputs to cover a branch, which triggers a dead-code failure report before any test runs. Sometimes, this is caused by a typo in the branch condition. For example, Keeper exposes such a failure in Verlan [72]. Verlan uses `object-detection` to judge whether an image contains an animal or not. Unfortunately, it wrongly uses `"animal"` instead of `obj.name == "animal"` in its branch condition, making the if-statement always True. It will regard every image that contains at least one object as an animal image!

```python
object = client.object_detection(image=img)
for obj in objects:
  if obj.name=="dog" or "animal":
    do_A()
```

Figure 8: Dead-code bugs in Verlan [72]

4.1.3 Generic failures. These have symptoms like crashes that do not require special techniques to observe. Comparing with traditional testing techniques, Keeper offers extra benefit in two scenarios. (1) The failures are caused by bugs located on a path that requires specific ML API inputs to trigger. Keeper contributes by generating the needed ML API inputs to exercise the path. (2) The failures are directly related to the corner cases of ML API inputs, such as blank images that cause `label_detection` to return no labels. An example of such a bug exposed by Keeper is illustrated in Figure 9.

```python
text = client.text_detection(image=img)
labels = text[0].description.split('\n')
for label in labels:
  do_something()
```

Figure 9: Crash failure in FortniteKillfeed [73]: a blank image returns an empty array `text` and trigger an index-out-of-range**.**

4.2 Failure attribution

To help developers understand and tackle accuracy failures, Keeper attempts to automatically patch the software by changing how ML APIs' output is used. Keeper suggests the change to developers and if all attempts failed, Keeper suggests developers to consider using a different, more accurate ML API, or adding extra input screening or pre-processing. Specifically, Keeper attempts two types of changes to the branch b where the failure is associated with.

Label changes. When branch b compares a ML API output with a set of labels, Keeper tries to expand the set of labels with three goals in mind. (1) Recall goal: more test inputs that are expected to exercise b can now satisfy b's condition; (2) Precision goal: most inputs that are not expected to exercise b should continue to fail the condition of b; (3) Semantic goal: the added labels are related to the original label(s) in b in terms of natural language semantics.

Without loss of generality, imagine that b takes the form of `if o == label0`, with `o` being the output of an ML API f. Keeper first collects the set of labels L output by f for every input in $\mathbb{I}_b^{\text{fail}}$, the set of inputs that are expected to exercise b but fail to do so.

Then, considering the semantic goal, Keeper filters out every label in L that is neither adjacent to nor sharing a common neighbor with `label0` in the wikidata knowledge graph [74]. For example, "amber" is pruned out by Keeper while processing the accuracy failure in Figure 1, because it is far away from "fire" in the knowledge graph. Instead, "flame" and "ash" both remain, as they are both adjacent to "fire" on the graph.

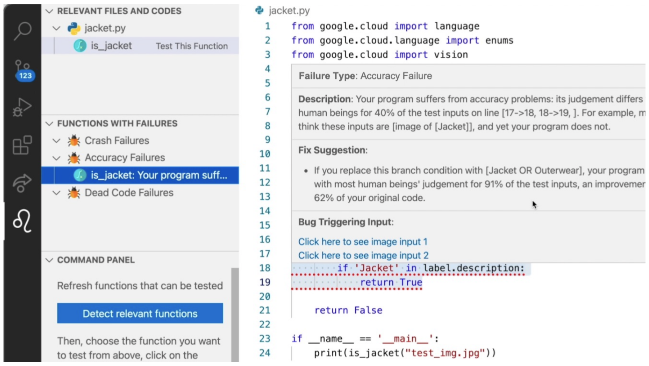
Next, Keeper uses a greedy algorithm to iteratively expand the set of labels compared with `o` in b. Every time, Keeper adds to the set a label `l` $\in L$ so that `l` offers the biggest improvement in b's recall without reducing b's F1-score (i.e., the harmonic mean of the precision and the recall). Here, the precision of branch b is computed as $\frac{|\mathbb{I}_b^{succ}|}{|\mathbb{I}^{suc}|}$: among all the inputs that exercise b, how many of them are expected to do so. This procedure continues until the recall of b goes above the accuracy failure threshold or when there is no eligible candidate label remaining in L.

Exactly through this process, Keeper suggests to the developers that the alarm branch in Figure 1 should check more labels like that in Figure 7, as by checking more labels the branch's recall can increase from 40% to 85% on those test cases generated by Keeper. This suggestion is proposed through a text description instead of a code patch—"If you additionally check flame and ash in the branch condition on Line 3, your program will agree with most human beings' judgement for 85% of test inputs, an improvement from 40% of your original code".

Threshold changes. As discussed earlier, an accuracy failure is reported when a branch b, which checks the `score` and/or `magnitude` output of a sentiment-detection API, gets exercised by many inputs labeled as having positive emotions and also many inputs labeled as having negative emotions. Keeper applies logistic regression to these input texts, with the {`score`, `magnitude`} output of each input as feature vectors and the labeled emotion as a class. Keeper then suggests the linear formula of logistic regression as a new branch checking threshold to developers, letting them know that this new formula can better differentiate text inputs with different emotions.

5 IMPLEMENTATION

We have implemented Keeper for Python applications that use Google Cloud AI APIs [1], the most popular cloud AI services on Github [7]. The core algorithm of Keeper is general to other languages and ML Cloud APIs. Keeper uses dynamic symbolic execution framework PyExZ3 [47], which implements the DSE algorithm, and uses CVC4 [75] for constraint solving. Keeper uses Python built-in trace back tool [76] to check branch coverage, and Pyan [77] and Jedi [78] for call graph and program dependency analysis. Keeper uses Python scikit-learn[79] library for linear regression models.



Refresh functions that can be tested

Then, choose the function you want to test from above, click on the

```
1   from google.cloud import language
2   from google.cloud.language import enums
3   from google.cloud import vision
```

Failure Type: Accuracy Failure

Description: Your program suffers from accuracy problems: its judgement differs human beings for 40% of the test inputs on line [17->18, 18->19,]. For example, m think these inputs are [image of [Jacket]], and yet your program does not.

Fix Suggestion:

- If you replace this branch condition with [Jacket OR Outerwear], your program with most human beings' judgement for 91% of the test inputs, an improvemer 62% of your original code.

Bug Triggering Input:

Click here to see image input 1
Click here to see image input 2

```
18          if 'Jacket' in label.description:
19              return True
20
21      return False
22
23  if __name__ == '__main__':
24      print(is_jacket("test_img.jpg"))
```

Figure 10: Keeper IDE plugin interface

We have implemented an IDE plugin for visualized interaction with Keeper, as the debugging and fixing of accuracy failures particularly requires developers' participation, as illustrated in Figure 10. The plugin is an extension in Visual Studio Code [46], a popular code editor supporting multiple languages. For any Python software, Keeper first identifies all functions that invoke ML APIs

Failure type	Root Cause	Related ML Task	Keeper	RReal	RReal+Noise	Fuzz.
Crash failures	Out-of-bound accesses	Text detection, entity detection	6	5	5	4
	Missing input validation*	Document classification	1	-	-	-
	Missing type conversion	-	1	1	1	1
Accuracy failures	Improper labels	Image classi., object detect., document classi.	9	-	-	-
	API limitations	Image classification, object detection	6	-	-	-
	Improper threshold	Sentiment detection	9	-	-	-
Dead-code failures	Typos	Image classification, text detection	2	-	-	-
	Non-existing label	Image classification	1	-	-	-

Table 2: Unique failures exposed by Keeper. (*: This crash disappeared later with the most recent version of Google API.)

directly or indirectly through callees, and displays them on the side bar, under “RELEVANT FILES AND CODES” in Figure 10. From that list, developers can select the function to test. Once they have made the selection, they will be asked to provide type information of function parameters, as Python is a dynamically typed language. Keeper will then start the testing. At the end of the testing, which usually takes 1–2 minutes, any execution failure that has been exposed is listed in the side bar, right under “FUNCTIONS WITH FAILURES” in the figure. Source code related to each failure is highlighted, together with a hovering window that offers detailed information like failure description, triggering inputs, and patch suggestions. A demo of the Keeper plugin can be found at our artifact [80].

6 EVALUATION

Our evaluation aims to answer several questions:

(1) Does Keeper help improve the branch coverage in testing?
(2) Is Keeper able to find bugs during its testing?
(3) Is Keeper able to suggest fixes for accuracy failures?

6.1 Methodology

6.1.1 Applications. We evaluate Keeper using 63 Python applications that are from two sources. 1) From the 360 open-source applications assembled by a previous study of ML APIs [7], we found 45 Python applications that use ML APIs in a non-trivial way (i.e., the API output affects control flow). 2) We additionally checked about 100 random Python applications on GitHub that use ML APIs and found 18 applications that use ML APIs in a non-trivial way.

These 63 applications use a range of ML APIs, including Vision (32 apps), Language (23 apps), and Speech (8). Their sizes range from 54 lines of code to more than 100,000 lines of code, with 582 lines of code being the median[3]. They have a median age of 18 months at the time of our study (*Apr. 1st, 2021*). Among these 63 applications, 16 applications have received 1 or multiple stars on Github; the other 47 applications have not received any stars. The details of each application, including the link to each Github code repository, are included in Table 4.

Despite our best effort in application collection, unfortunately, most of these 63 applications seem to be research projects, hackathon products, or demo programs, based on their limited popularity in Github. This is probably due to the young age of ML APIs. Consequently, our evaluation results may not generalize to mature software that has a solid user base.

	Vision App.	Language App.	Speech App.
Keeper	91.9%	91.5%	89.7%
Random-Real	74.5%	85.0%	54.3%
Random-Real-Noise	73.0%	65.2%	54.3%
Fuzzing	44.4%	74.0%	24.9%

Table 3: Average branch coverage across 63 applications.

For more than half of the applications (35), we simply specify `main` as the function to test. In other cases, the function under test is the entry function to the software feature related to ML APIs. The average number of branches in these functions-to-test is 13.

6.1.2 Baselines. We compare Keeper with 3 other techniques. Each technique generates 100 test inputs for each function under test.

(1) *Random Real*: we randomly pick inputs from well established data sets, including ImageNet [67] that contains 14 million images, Twitter US Airline Sentiment [81] that contains 15,000 tweets, and a set of audio clips synthesized for 115 daily sentences [82].

(2) *Random Real + Noise*: we add random noise to inputs picked by *Random Real*. For an image, we randomly added noises following Gaussian distribution; for an text input, we randomly decide whether to add noise and if so, randomly changed the word orders. For audio input, we do not add noise here, as we found that adding small noises does not affect ML API and yet adding big noises would turn the audio clip into what the third approach will generate.

(3) *Fuzzing*: we use a coverage-based fuzzing tool pythonfuzz [83] to generate images, text, and audio. For every image input, we use an integer list to fill its RGB matrix in a repeated way. For every text inputs, we generates ASCII character sequences. For audio inputs, we directly generates the audio data.

6.2 Software testing evaluation

6.2.1 Branch coverage. For each of the 63 functions specified to test, each from one application in our benchmark suite, we compute the accumulative branch coverage achieved by the 100 inputs generated by each testing technique. Table 3 shows the overall results.

Across different types of applications, Keeper consistently achieves high branch coverage, around 90% on average. The uncovered branches are either related to dead-code failures that Keeper discovers, or related to code that our underlying symbolic execution engine cannot handle. In comparison, the fuzzing technique performed the worst, covering less than 50% of the branches for vision and speech applications, confirming our intuition that it is important to use realistic inputs to test ML APIs.

[3] Files from templates, frameworks, and libraries are not included in the LoC counting.

Application w/ link	Description	Stars	LOC	Branch Coverage					Keeper Exposed Failures		
				#Branches	Keeper	RReal	Rreal+Noise	Fuzz	Accuracy	Crash	Dead-code
selfmailbot	Telegram bot	114	911	10	90%	70%	10%	10%			
FortniteKillfeed	Game assistant	5	1440	28	100%	0%	0%	0%		1	
FB_MMHM	Meme inspector	3	2850	34	94%	88%	91%	53%			
Audio-SentenceSplit	Audio splitter	3	304	4	100%	50%	50%	50%			
calbot	Nutrition tracker	2	653	8	100%	100%	100%	25%			
Hapi	Produce analyzer	2	261	12	100%	100%	92%	92%			
stockmine	Investment helper	2	1079	10	90%	80%	70%	50%	1		
Tone	Smart music player	2	12709	4	100%	100%	25%	25%	1		
UOttaHack-2019	Speech emotion detector	2	107	6	100%	100%	83%	83%	1		
BlindHandAssistance	Blind assistant	2	708	22	27%	18%	18%	14%			
IngredientPrediction	Recipe recommender	1	142	12	100%	58%	58%	58%	3		
recipeGO	Recipe recommender	1	22457	4	100%	100%	100%	25%			
devfest	Public opinion analyzer	1	206	10	100%	80%	50%	80%	1		
Klassroom	Note taker	1	17250	14	100%	100%	100%	86%			
uofthacks6	News summerizer	1	495	48	96%	79%	79%	75%		1	
HackThe6ix	Insurance manager	1	65944	46	87%	72%	70%	63%			
Average branch coverage / Total failures exposed by Keeper					**93%**	**75%**	**62%**	**49%**	**7**	**2**	**0**
Aander-ETL	Smart album	0	471	16	81%	75%	81%	63%	3		
Alpr	License recognization	0	89	4	100%	75%	75%	50%			
artificial_intelligence	Calorie calculator	0	401	26	81%	35%	35%	4%			
emotion2music	Smart music player	0	777	10	100%	70%	70%	10%			
Experiments	Product info analyzer	0	2500	18	100%	100%	100%	6%			
FESMKMITL	Emotion tagger	0	1024	8	63%	63%	63%	63%			1
heapsortcypher	Garbage classifier	0	85	12	100%	83%	92%	75%	3		
Image-analyzer-chat-bot	Chat bot	0	163	12	100%	100%	92%	33%			
ns_online_toolkit	Game assistant	0	9907	8	100%	88%	88%	25%			
Phoenix	Fire alarm	0	284	6	100%	83%	83%	83%	1		
ResearchSpring2019	Prescription reader	0	131065	62	35%	0%	0%	0%		1	
SeeFarBeyond	Coin finder	0	280	54	70%	39%	39%	39%	2	2	
smart_can	Garbage classifier	0	1750	14	100%	100%	100%	79%			
SnapCal	Smart calendar	0	233	4	75%	75%	75%	50%			
twimage-search	Landmark recognizer	0	107	26	100%	100%	88%	46%			
WanderStub	Exchange convertor	0	54	2	100%	0%	0%	0%		1	
Verlan	Animal finder	0	73	4	75%	75%	75%	25%			1
thgml	Calorie calculator	0	76476	8	100%	38%	38%	13%			
SBHacks2021	Smart camera	0	234	10	100%	100%	100%	70%			
flood_depths	Flood monitor	0	203	4	100%	100%	100%	50%			
image_tagging	Fruit checker	0	749	4	100%	100%	100%	100%			
shecodes-hack	Clothes checker	0	7052	4	100%	75%	100%	75%			
SunHacks2019	Blind assistant	0	40071	14	100%	100%	79%	7%	1		
SnapTrack_HACK112	Nutrition tracker	0	1096	4	100%	100%	100%	100%			1
lahacks-quaranteen	Image checker	0	4080	2	100%	100%	50%	50%			
plant-watcher	Plant manager	0	183	8	75%	75%	75%	75%	1		
senior-project	Smart album	0	582	10	90%	90%	90%	70%			
animal_analysis	Image sharing platform	0	355	10	90%	80%	60%	50%			
calhacksv2	Movie review analyzer	0	17105	10	100%	100%	90%	80%			
carbon-hack-sentiment	Public opinion analyzer	0	222	8	100%	100%	88%	88%	1		
Cloud-Computing	Food delivery	0	26914	4	100%	100%	25%	100%	1		
EC601_twitter_keyword	Investment helper	0	2563	6	100%	100%	83%	83%	1		
ElectionSentimentAnalysis	Tweet analyzer	0	1801	8	100%	100%	88%	100%			
GeoScholar	Scholar database	0	268	4	100%	100%	100%	100%		1	
JournalBot	Journel manager	0	295	6	100%	100%	83%	100%	1		
noteScript	Note taker	0	886	16	88%	44%	44%	44%	1	1	
Sarcatchtic-MakeSPP19	Text tone checker	0	259	8	100%	100%	100%	88%	1		
Twitter_Mining_GAE	Disaster news analyzer	0	822	6	67%	67%	67%	67%			
BadGIF	Discord bot	0	19747	4	100%	50%	50%	75%			
Mind_Reading_Journal	Journel manager	0	558	6	100%	100%	67%	83%			
newsChronicle	Timeline generator	0	346	4	100%	100%	100%	100%			
ocr-contratos	Contract analyzer	0	22931	6	83%	83%	67%	67%			
most_anoying_app_ever	Smart music player	0	176	8	100%	63%	63%	38%			
PottyPot	Swear remover	0	135	14	100%	71%	71%	14%			
ReadingMachine	Book reader	0	209	4	100%	75%	75%	25%			
SwearRemoval	Swear remover	0	203	10	100%	30%	30%	20%			
TRANSLATOR	Consecutive interpreter	0	10700	14	100%	57%	57%	29%			
Average branch coverage / Total failures exposed by Keeper					**93%**	**78%**	**72%**	**55%**	**17**	**6**	**3**

Table 4: Information and results of 63 applications. (Each application name contains a hyper link to its GitHub repository; the LOC numbers refer only to the actual application code, not libraries, templates, or other files in the repository; #Branches: the number of branches in the function under test; 0% coverage: all test cases crash the program execution before reaching any branches.)

Random Real performs better than fuzzing, but still fails to cover about a quarter of branches in vision applications and half of the branches in speech applications. Adding random noises to random realistic inputs does not help. Keeper covers 23% and 59% more branches than Random-Real for vision and speech applications, respectively, as Keeper leverages symbolic execution and pseudo-inverse functions to generate inputs targeting different branches.

Applications that use language APIs appear to be the easiest to cover—even fuzzing achieves 74% coverage. This is probably because language APIs’ output, like document type or entity name, has much less variation than that of vision and speech APIs.

Table 4 shows the exact branch coverage offered by each technique for each benchmark application. As we can see, Keeper offers the highest branch coverage for all 63 applications.

6.2.2 Failure exposing and attribution. As shown in Table 2, Keeper exposed many failures by running those 100 test inputs it generated: 35 failures from the latest version of 25 applications. These failures cover a range of symptoms and root causes. Except for one failure caused by missing type conversion, the others are all related to different types of cognitive ML tasks, as shown in the table.

In comparison, alternative testing techniques missed 2–3 crash failures caught by Keeper. Furthermore, unlike Keeper, they cannot automatically recognize accuracy failures and dead-code failures.

Accuracy failures. Among the 24 accuracy failures exposed by Keeper, 15 of them are related to label checking for vision APIs and document-classification API, and 9 are related to threshold checking for the sentiment detection API.

For all of the 9 failures related to sentiment detection, Keeper manages to suggest better checking threshold that fixes the failure.

There are 9 accuracy failures that Keeper manages to fix by making the failure branch check for 1–3 extra labels. The failure in Figure 1 is one such example. As another example, one application checks if the output of `label_detection` contains either “building” or “estate” or “mansion”. This branch’s recall is very low: 33%. Keeper suggests adding “house”, “architecture”, and “window” to the label set, which would improve the recall to be above 75%.

For the remaining 6 vision-related accuracy failures, code changes by Keeper can alleviate the problem but cannot push the recall of the related branch to be above 75%, suggesting fundamental API limitations. Two of these cases actually involve non-existing labels. For example, the “aluminum” in line 7 of Figure 4 is actually a non-existing label. Keeper suggests checking “metal” instead, which increases the branch’s recall to close to 40%, but still below 75%.

Deadcode failures occurred in 3 applications. One of them is due to non-existing labels. Two are because of typos in branches that process ML API output, like the one in Figure 8.

Crash failures are mainly caused by out-of-bound accesses to lists returned by ML APIs, as shown in Figure 9. One crash is caused by buggy code inside a branch body that handles images with coins inside. This failure cannot be exposed by other testing techniques, as they did not produce images with coins inside.

False positives. Keeper has two false positives in total (they are not included in Table 2). One application tries to detect sensitive document by checking if any output of the document-classification API contains a “ensitive” sub-string. Keeper feeds its pseudo-inverse function with “ensitive” and fails to get any test inputs, and hence incorrectly reports a dead-code failure. The other application has a branch that gets covered only when an ML API generates a specific output with low confidence. Keeper is not effective at generating low-confidence inputs and wrongly reports an accuracy failure.

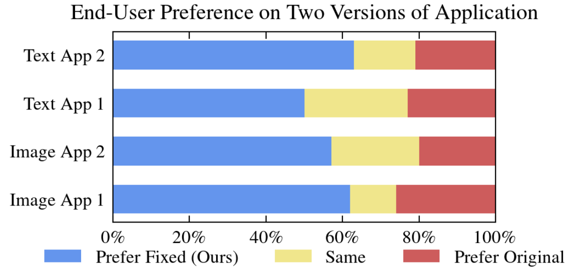


Figure 11: End-user preference: Original vs. Keeper version.

Threshold setting. As discussed in Section 4.1, the recall threshold α is set to 0.75 by default when detecting accuracy failures. Naturally, more failures would be reported when α is larger. Increasing α to 0.95, which is unreasonably high, would creates 5 more failure reports; decreasing α to 0.6 would have 2 fewer failure reports.

Results across applications. As shown in Table 4, the 35 failures exposed by Keeper are from 25 different applications. These include both applications that have received stars on Github and those that have not; both the smallest application in our benchmark suite, WanderStub, and the largest one, ResearchSpring2019.

6.3 User studies

To better evaluate the accuracy failures and the code changes suggested by Keeper, we recruited 100 participants on Amazon Mechanical Turk (Mturk) for a software-user survey. The survey includes 4 applications from our benchmark suites: 2 image-related applications and 2 text-related applications. On each survey page, a brief description is given for an application and user-study participants are told to review how two versions of this application perform on a set of inputs. Then, the web page displays a number of input images/text and the corresponding outputs of application version-1 and application version-2. These two versions are the original application and the application with suggested code changes from Keeper (referred to as *fixed* in Figure 11); we randomly decide which one of them is version-1 and which is version-2 on each survey page to reduce potential bias. Each participant is asked to answer questions about (1) for each input, which version’s output they prefer; and (2) which version they think is better with everything considered. Participants were compensated $5 after the survey.

A summary of the user study results is shown in Figure 11. As we can see, in all cases, a dominate portion of end-users prefer the version with changes suggested by Keeper over the original version, supporting Keeper’s judgement about accuracy failures and Keeper’s attempt in fixing the accuracy problems. At the same time, we also noticed that there are 20–26% of user-study participants who prefer the original software and 12–27% who feel the two versions are about the same. These results confirm the fact that cognitive tasks are inherently subjective—even human beings often do not agree with each other on these tasks.

7 THREATS TO VALIDITY

Internal threats to validity. Keeper assumes that search engines' top results are mostly consistent with human judgement, which could be incorrect. The failure identification and fixing attempts in Keeper are inherently probabilistic. The recall that Keeper calculated for each branch could vary depending on the test inputs. More test inputs would make the testing procedure more robust.

Some inputs generated by Keeper may not be the inputs that the software aims to handle, like the image being a photo taken indoor and yet the software meant to be used outdoor. When Keeper expands a branch's comparison label set, the increase of the recall sometimes comes with the decrease of the precision (i.e., more inputs not expected to exercise the branch does exercise). Although Keeper uses the F1-score to balance precision and recall, ultimately developers need to make the code change decision. We implemented Keeper IDE plug-in, aiming to help developers make informed decision about how their software uses ML APIs.

When an input expected by Keeper to cover a branch b fails to do so, this input may cover another branch b' whose body conducts the same computation as b. This would confuse Keeper's failure identification, although we have not observed such situations.

External threats to validity. Most of applications in our benchmark suite, including those used as examples in the paper, are research applications, hackathon projects, or demo programs. Consequently, observations and results obtained from them might not generalize to more widely used, real-world applications. Our tool is only tested with python applications using Google AI, not other ML Cloud API services.

8 RELATED WORK

ML-related software. Prior work studied development phases [5, 84–87] of software that contains machine learning components. They do not look at how to test such software.

A recent study manually identified anti-patterns [7] from software that uses ML APIs. Keeper differs from this study by proposing testing techniques that can *automatically* expose failures and attribute failure causes. On the contrary, this recent study obtained all its anti-patterns through manual code inspection. It managed to build automated detectors for some performance-related anti-patterns, like repeatedly calling a ML API with a constant input, but does not have automated bug detection or testing solutions for any correctness-related anti-patterns. Furthermore, due to the different design goals, the type of failure root causes covered by Keeper also differs from the previous study. In the 45 applications that are evaluated both by Keeper and the previous study, Keeper automatically exposed 32 failures, among which only 3 were also identified by the previous study.

Another line of work [88–91] studies testing autonomous systems. They are tailored for the characteristics of autonomous driving and spatial-temporal data, and thus not applicable to most ML software targeted by Keeper.

ML-related testing. Much research has been done for testing [9, 17–41, 92] and fixing [42–45] neural networks, in terms of accuracy, fairness, and security. Other work studies implementation bugs of neural network architectures [93, 94] and other machine learning models [95, 96]. They are orthogonal to Keeper.

As discussed in Section 1, some previous work looked at how to test specific software that contains ML components [12–15]. Unfortunately, their solutions do not apply to general ML software. For example, one work trained a SVM classifier to judge the correctness of an image dilation program, leveraging the fact that the input image and the output image should contain the same objects [12]. To test a blood-vessel image categorizer, previous work [13] generates blood-vessel images with certain density, branches, and other features, and use these features to generate output ground truth. Previous work [14, 15] uses metamorphic approaches to test entity detection and image region growth programs. They require application-specific rules about inputs and outputs relationship (e.g., after we concatenate inputs of entity detection, the output becomes the concatenation of individual outputs [15]).

Prior work studies automatic testing and bug detection of machine learning APIs, including frameworks for implementing neural networks [97–103] and REST APIs that provide machine learning solutions [104–106]. They focus on the implementation inside ML APIs, not how they interact with other software components.

Test generation using search engines. Previous work [107, 108] explored using search engines to generate string inputs for software under test. Specifically, when a program identifier corresponds to a common concept, such as `emailAddress`, this identifier can be used as a keyword to search for related web pages. The resulting web pages can then be processed to help generate related string inputs (e.g., a realistic email address). Clearly, Keeper tackles fundamentally different problems from previous work, although Keeper also leverages search engines.

9 CONCLUSION

It is challenging to efficiently and effectively test software containing machine learning components. We present Keeper, an automated coverage-guided testing framework that helps developers to detect bugs and provide fixing suggestions for their software implementation. Keeper automatically generates test cases via a novel two-stage symbolic execution and Keeper-designed ML inverse functions. We evaluate Keeper with a variety of open-source machine learning applications and achieve high code coverage with a small set of test cases. It identifies bugs that leads to software crash, lower inference accuracy, or dead code.

10 DATA AVAILABILITY

We release our benchmarks, the tool source code, experimental results, and user study results online [80].

ACKNOWLEDGEMENT

We thank the reviewers for their insightful feedback. The authors' research is supported by NSF (CNS1764039, CNS1956180, CCF1837120, CCF2119184, CNS1952050, CCF1823032), ARO (W911NF1920321), and a DOE Early Career Award (grant DESC0014195 0003). Additional support comes from the CERES Center for Unstoppable Computing, CDAC Summer Lab, the Marian and Stuart Rice Research Award, Microsoft research dissertation grant, UChicago College Research Fellow Grant, and research gifts from Facebook.

REFERENCES

[1] Google, “Google cloud ai.” Online document https://cloud.google.com/products/ai, 2020.

[2] Amazon, “Amazon artificial intelligence service.” Online document https://aws.amazon.com/machine-learning/ai-services, 2020.

[3] Microsoft, “Microsoft azure cognitive services.” Online document https://azure.microsoft.com/en-us/services/cognitive-services, 2020.

[4] IBM, “Ibm watson.” Online document https://www.ibm.com/watson, 2020.

[5] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann, “Software engineering for machine learning: A case study,” in *ICSE-SEIP*, pp. 291–300, IEEE, 2019.

[6] K. Das and R. N. Behera, “A survey on machine learning: concept, algorithms and applications,” *International Journal of Innovative Research in Computer and Communication Engineering*, vol. 5, no. 2, pp. 1301–1309, 2017.

[7] C. Wan, S. Liu, H. Hoffmann, M. Maire, and S. Lu, “Are machine learning cloud apis used correctly?,” in *43th International Conference on Software Engineering (ICSE’21)*, 2021.

[8] Phoenix, “A fire-detection application.” https://github.com/yunusemreemik/Phoenix.

[9] A. Odena, C. Olsson, D. Andersen, and I. Goodfellow, “Tensorfuzz: Debugging neural networks with coverage-guided fuzzing,” in *ICML*, 2019.

[10] X. Xie, L. Ma, F. Juefei-Xu, H. Chen, M. Xue, B. Li, Y. Liu, J. Zhao, J. Yin, and S. See, “Deephunter: Hunting deep neural network defects via coverage-guided fuzzing,” *arXiv preprint arXiv:1809.01266*, 2018.

[11] P. Ma, S. Wang, and J. Liu, “Metamorphic testing and certified mitigation of fairness violations in nlp models.,” in *IJCAI*, pp. 458–465, 2020.

[12] T. Jameel, L. Mengxiang, and L. Chao, “Automatic test oracle for image processing applications using support vector machines,” in *2015 6th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, pp. 1110–1113, IEEE, 2015.

[13] M. C. Júnior, R. A. Oliveira, M. A. Valverde, M. P. Jackowski, F. L. Nunes, and M. E. Delamaro, “Feature-based test oracles to categorize synthetic 3d and 2d images of blood vessels,” in *Proceedings of the 2nd Brazilian Symposium on Systematic and Automated Software Testing*, pp. 1–6, 2017.

[14] C. Jiang, S. Huang, and Z. Hui, “Metamorphic testing of image region growth programs in image processing applications,” in *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 2018.

[15] M. Srinivasan, M. P. Shahri, I. Kahanda, and U. Kanewala, “Quality assurance of bioinformatics software: a case study of testing a biomedical text processing tool using metamorphic testing,” in *Proceedings of the 3rd International Workshop on Metamorphic Testing*, pp. 26–33, 2018.

[16] J. Wang, G. Dong, J. Sun, X. Wang, and P. Zhang, “Adversarial sample detection for deep neural network through model mutation testing,” in *ICSE*, 2019.

[17] K. Pei, Y. Cao, J. Yang, and S. Jana, “Deepxplore: Automated whitebox testing of deep learning systems,” in *ASPLOS*, 2017.

[18] Y. Tian, K. Pei, S. Jana, and B. Ray, “Deeptest: Automated testing of deep-neural-network-driven autonomous cars,” in *ICSE*, 2018.

[19] X. Xie, L. Ma, F. Juefei-Xu, M. Xue, H. Chen, Y. Liu, J. Zhao, B. Li, J. Yin, and S. See, “Deephunter: A coverage-guided fuzz testing framework for deep neural networks,” in *ISSTA*, pp. 146–157, 2019.

[20] S. Ma, Y. Liu, W.-C. Lee, X. Zhang, and A. Grama, “Mode: automated neural network model debugging via state differential analysis and input selection,” in *ESEC/FSE*, 2018.

[21] S. Ma, Y. Aafer, Z. Xu, W.-C. Lee, J. Zhai, Y. Liu, and X. Zhang, “Lamp: data provenance for graph based machine learning algorithms through derivative computation,” in *FSE*, 2017.

[22] N. D. Bui, Y. Yu, and L. Jiang, “Autofocus: interpreting attention-based neural networks by code perturbation,” in *ASE*, 2019.

[23] R. B. Abdessalem, S. Nejati, L. C. Briand, and T. Stifter, “Testing vision-based control systems using learnable evolutionary algorithms,” in *ICSE*, 2018.

[24] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao, *et al.*, “Deepmutation: Mutation testing of deep learning systems,” in *ISSRE*, 2018.

[25] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid, “Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems,” in *ASE*, 2018.

[26] A. Dwarakanath, M. Ahuja, S. Sikand, R. M. Rao, R. J. C. Bose, N. Dubash, and S. Podder, “Identifying implementation bugs in machine learning based image classifiers using metamorphic testing,” in *ISSTA*, 2018.

[27] S. Galhotra, Y. Brun, and A. Meliou, “Fairness testing: testing software for discrimination,” in *FSE*, 2017.

[28] R. Angell, B. Johnson, Y. Brun, and A. Meliou, “Themis: Automatically testing software for discrimination,” in *ESEC/FSE*, 2018.

[29] S. Amershi, M. Chickering, S. M. Drucker, B. Lee, P. Simard, and J. Suh, “Modeltracker: Redesigning performance analysis tools for machine learning,” in *CHI*, 2015.

[30] S. Yan, G. Tao, X. Liu, J. Zhai, S. Ma, L. Xu, and X. Zhang, “Correlations between deep neural network model coverage criteria and model quality,” in *ESEC/FSE*, 2020.

[31] F. Zhang, S. P. Chowdhury, and M. Christakis, “Deepsearch: A simple and effective blackbox attack for deep neural networks,” in *ESEC/FSE*, 2020.

[32] F. Harel-Canada, L. Wang, M. A. Gulzar, Q. Gu, and M. Kim, “Is neuron coverage a meaningful measure for testing deep neural networks?,” in *ESEC/FSE*, 2020.

[33] V. Riccio and P. Tonella, “Model-based exploration of the frontier of behaviours for deep learning system testing,” in *ESEC/FSE*, 2020.

[34] S. Gerasimou, H. F. Eniser, A. Sen, and A. Cakan, “Importance-driven deep learning system testing,” in *ICSE*, 2020.

[35] B. Paulsen, J. Wang, and C. Wang, “Reludiff: Differential verification of deep neural networks,” in *ICSE*, 2020.

[36] X. Zhang, X. Xie, L. Ma, X. Du, Q. Hu, Y. Liu, J. Zhao, and M. Sun, “Towards characterizing adversarial defects of deep learning software from the lens of uncertainty,” in *ICSE*, 2020.

[37] D. Berend, X. Xie, L. Ma, L. Zhou, Y. Liu, C. Xu, and J. Zhao, “Cats are not fish: Deep learning testing calls for out-of-distribution awareness,” in *FSE*, 2020.

[38] Y. Feng, Q. Shi, X. Gao, J. Wan, C. Fang, and Z. Chen, “Deepgini: prioritizing massive tests to enhance the robustness of deep neural networks,” in *ISSTA*, 2020.

[39] S. Dutta, A. Shi, R. Choudhary, Z. Zhang, A. Jain, and S. Misailovic, “Detecting flaky tests in probabilistic and machine learning applications,” in *ISSTA*, 2020.

[40] S. Lee, S. Cha, D. Lee, and H. Oh, “Effective white-box testing of deep neural networks with adaptive neuron-selection strategy,” in *ISSTA*, 2020.

[41] A. Sharma and H. Wehrheim, “Higher income, larger loan? monotonicity testing of machine learning models,” in *ISSTA*, 2020.

[42] H. Zhang and W. Chan, “Apricot: a weight-adaptation approach to fixing deep learning models,” in *ASE*, 2019.

[43] Z. Li, X. Ma, C. Xu, J. Xu, C. Cao, and J. Lü, “Operational calibration: Debugging confidence errors for dnns in the field,” in *ESEC/FSE*, 2020.

[44] Z. Sun, J. M. Zhang, M. Harman, M. Papadakis, and L. Zhang, “Automatic testing and improvement of machine translation,” in *ICSE*, 2020.

[45] M. J. Islam, R. Pan, G. Nguyen, and H. Rajan, “Repairing deep neural networks: Fix patterns and challenges,” in *ICSE*, 2020.

[46] Microsoft, “Visual studio code.” Online document https://code.visualstudio.com/, 2021.

[47] M. Irlbeck *et al.*, “Deconstructing dynamic symbolic execution,” *Dependable Software Systems Engineering*, vol. 40, p. 26, 2015.

[48] H. Pham, Z. Dai, Q. Xie, and Q. V. Le, “Meta pseudo labels,” in *CVPR*, 2021.

[49] recipeGo, “A recipe recommendation application.” https://github.com/Reckonzz/recipeGO.

[50] emotion2music, “A smart music player application.” https://github.com/varnachandar/emotion2music.

[51] NsTool, “A monitor application.” https://github.com/clarkwkw/ns_online_toolkit.

[52] noteScript, “A lecture note application.” https://github.com/GalenWong/noteScript.

[53] stockmine, “A stock prediction application.” https://github.com/nicholasadamou/stockmine.

[54] A. Go, R. Bhayani, and L. Huang, “Twitter sentiment classification using distant supervision,” *CS224N project report, Stanford*, vol. 1, no. 12, p. 2009, 2009.

[55] Klassroom, “A lecture note application.” https://github.com/dev5151/Klassroom.

[56] TRANSLATOR, “A smart light application.” https://github.com/mubeenafatima/TRANSLATOR.

[57] HeapSortCypher, “A garbage classification application.” https://github.com/matthew-chu/heapsortcypher.

[58] D. Chaffey, “Search engine marketing statistics 2020.” https://www.smartinsights.com/search-engine-marketing/search-engine-statistics/.

[59] M. Bing, “Bing image search.” https://www.bing.com/images/trending?FORM=ILPTRD.

[60] “Lorem picsum.” https://picsum.photos.

[61] “Wikipedia.” https://en.m.wikipedia.org/.

[62] “Encyclopedia britannica.” https://www.britannica.com/.

[63] “Pillow: Python imaging library.” https://pypi.org/project/Pillow/.

[64] WanderStub, “An exchange conversion application.” https://github.com/richardjpark26/WanderStub.

[65] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, “Language models are unsupervised multitask learners,” *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[66] “pyttsx3: Text-to-speech library for python.” https://pypi.org/project/pyttsx3/.

[67] J. Deng, W. Dong, R. Socher, L. Li, K. Li, and L. Fei-Fei, “Imagenet: A large-scale hierarchical image database,” in *CVPR*, 2009.

[68] A. Kuznetsova, H. Rom, N. Alldrin, J. Uijlings, I. Krasin, J. Pont-Tuset, S. Kamali, S. Popov, M. Malloci, A. Kolesnikov, *et al.*, “The open images dataset v4,” *International Journal of Computer Vision*, pp. 1–26, 2020.

[69] J. Gui, Z. Sun, Y. Wen, D. Tao, and J. Ye, “A review on generative adversarial networks: Algorithms, theory, and applications,” *arXiv preprint arXiv:2001.06937*, 2020.

[70] A. Ramesh, M. Pavlov, G. Goh, S. Gray, C. Voss, A. Radford, M. Chen, and I. Sutskever, "Zero-shot text-to-image generation," *arXiv preprint arXiv:2102.12092*, 2021.

[71] FESMKMITL, "A smart camera application." https://github.com/matthewjmc/FESMKMITL.

[72] Verlan, "A pet application." https://github.com/sarvesh-tech/Verlan.

[73] FortniteKillfeed, "A real time tracker application." https://github.com/Godsinred/FortniteKillfeed.

[74] "Wikipedia." https://www.wikidata.org/.

[75] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings* (G. Gopalakrishnan and S. Qadeer, eds.), vol. 6806 of *Lecture Notes in Computer Science*, pp. 171–177, Springer, 2011.

[76] "Python system-specific parameters and functions." https://docs.python.org/3/library/sys.html#sys.settrace.

[77] D. Marby and N. Yonskai, "Pyan3: Offline call graph generator for python 3." https://github.com/davidfraser/pyan.

[78] D. Halter, "Jedi: an awesome auto-completion, static analysis and refactoring library for python." Online document https://jedi.readthedocs.io.

[79] "scikit-learn: Machine learning in python." https://scikit-learn.org/stable/.

[80] C. Wan, S. Liu, S. Xie, Y. Liu, H. Hoffmann, M. Maire, and S. Lu, "Project Webpage: Accurate Learning for EneRgy and Timeliness in Software System." https://alert.cs.uchicago.edu/#release.

[81] Kaggle, "Twitter us airline sentiment." https://www.kaggle.com/crowdflower/twitter-airline-sentiment.

[82] "100 english daily sentences for daily use." https://englishspeakingcourse.net/100-english-sentences-for-daily-use/.

[83] Fuzzit.dev, "Pythonfuzz: coverage-guided fuzz testing for python." https://gitlab.com/gitlab-org/security-products/analyzers/fuzzers/pythonfuzz.

[84] C. Hill, R. Bellamy, T. Erickson, and M. Burnett, "Trials and tribulations of developers of intelligent systems: A field study," in *VL/HCC*, 2016.

[85] M. Kim, T. Zimmermann, R. DeLine, and A. Begel, "The emerging role of data scientists on software development teams," in *ICSE*, 2016.

[86] M. Kim, T. Zimmermann, R. DeLine, and A. Begel, "Data scientists in software teams: State of the art and challenges," *TSE*, 2017.

[87] X. Zhao and X. Gao, "An ai software test method based on scene deductive approach," in *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pp. 14–20, IEEE, 2018.

[88] P. Helle, W. Schamai, and C. Strobel, "Testing of autonomous systems–challenges and current state-of-the-art," in *INCOSE international symposium*, 2016.

[89] T. Linz, "Testing autonomous systems," in *The Future of Software Quality Assurance*, pp. 61–75, Springer, Cham, 2020.

[90] M. Lindvall, A. Porter, G. Magnusson, and C. Schulze, "Metamorphic model-based testing of autonomous systems," in *2017 IEEE/ACM 2nd International Workshop on Metamorphic Testing (MET)*, 2017.

[91] H. Khosrowjerdi and K. Meinke, "Learning-based testing for autonomous systems using spatial and temporal requirements," in *Proceedings of the 1st International Workshop on Machine Learning and Software Engineering in Symbiosis*, 2018.

[92] H. Zhu, D. Liu, I. Bayley, R. Harrison, and F. Cuzzolin, "Datamorphic testing: A method for testing intelligent applications," in *2019 IEEE International Conference On Artificial Intelligence Testing (AITest)*, 2019.

[93] Y. Zhang, L. Ren, L. Chen, Y. Xiong, S.-C. Cheung, and T. Xie, "Detecting numerical bugs in neural network architectures," in *ESEC/FSE*, 2020.

[94] G. Jahangirova, N. Humbatova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, "Taxonomy of real faults in deep learning systems," in *ICSE*, 2020.

[95] Y. Tao, S. Tang, Y. Liu, Z. Xu, and S. Qin, "How do api selections affect the runtime performance of data analytics tasks?," in *ASE*, 2019.

[96] D. Cheng, C. Cao, C. Xu, and X. Ma, "Manifesting bugs in machine learning code: An explorative study with mutation testing," in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pp. 313–324, IEEE, 2018.

[97] H. V. Pham, T. Lutellier, W. Qi, and L. Tan, "Cradle: cross-backend validation to detect and localize bugs in deep learning libraries," in *ICSE*, 2019.

[98] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, *et al.*, "Api design for machine learning software: experiences from the scikit-learn project," *arXiv preprint arXiv:1309.0238*, 2013.

[99] E. R. Sparks, A. Talwalkar, V. Smith, J. Kottalam, X. Pan, J. Gonzalez, M. J. Franklin, M. I. Jordan, and T. Kraska, "Mli: An api for distributed machine learning," in *ICDM*, 2013.

[100] S. Bahrampour, N. Ramakrishnan, L. Schott, and M. Shah, "Comparative study of deep learning software frameworks," *arXiv preprint arXiv:1511.06435*, 2015.

[101] M. Nejadgholi and J. Yang, "A study of oracle approximations in testing deep learning libraries," in *ASE*, 2019.

[102] Q. Guo, X. Xie, Y. Li, X. Zhang, Y. Liu, L. Xiaohong, and C. Shen, "Audee: Automated testing for deep learning frameworks," in *FSE*, 2020.

[103] S. Tizpaz-Niari, P. Cernỳ, and A. Trivedi, "Detecting and understanding real-world differential performance bugs in machine learning libraries," in *ISSTA*, 2020.

[104] F. Petrillo, P. Merle, N. Moha, and Y.-G. Guéhéneuc, "Are rest apis for cloud computing well-designed? an exploratory study," in *ICSOC*, pp. 157–170, Springer, 2016.

[105] E. Gossett, C. Toher, C. Oses, O. Isayev, F. Legrain, F. Rose, E. Zurek, J. Carrete, N. Mingo, A. Tropsha, *et al.*, "Aflow-ml: A restful api for machine-learning predictions of materials properties," *Computational Materials Science*, 2018.

[106] P. Godefroid, D. Lehmann, and M. Polishchuk, "Differential regression testing for rest apis," in *ISSTA*, 2020.

[107] P. McMinn, M. Shahbaz, and M. Stevenson, "Search-based test input generation for string data types using the results of web queries," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pp. 141–150, IEEE, 2012.

[108] M. Shahbaz, P. McMinn, and M. Stevenson, "Automatic generation of valid and invalid test data for string validation routines using web searches and regular expressions," *Science of Computer Programming*, vol. 97, pp. 405–425, 2015.