Keeper: Automated Testing and Fixing of Machine Learning Software

CHENGCHENG WAN, East China Normal University, China
SHICHENG LIU, Stanford University, United States
SOPHIE XIE, University of California, Berkeley, United States
YUHAN LIU, University of Chicago, United States
HENRY HOFFMANN, University of Chicago, United States
MICHAEL MAIRE, University of Chicago, United States
SHAN LU, Microsoft Research and University of Chicago, United States

The increasing number of software applications incorporating machine learning (ML) solutions has led to the need for testing techniques. However, testing ML software requires tremendous human effort to design realistic and relevant test inputs, and to judge software output correctness according to human common sense. Even when misbehavior is exposed, it is often unclear whether the defect is inside ML API or the surrounding code, and how to fix the implementation. This article tackles these challenges by proposing Keeper, an automated testing and fixing tool for ML software.

The core idea of Keeper is designing pseudo-inverse functions that semantically reverse the corresponding ML task in an empirical way and proxy common human judgment of real-world data. It incorporates these functions into a symbolic execution engine to generate tests. Keeper also detects code smells that degrade software performance. Once misbehavior is exposed, Keeper attempts to change how ML APIs are used to alleviate the misbehavior.

Our evaluation on a variety of applications shows that Keeper greatly improves branch coverage, while identifying 74 previously unknown failures and 19 code smells from 56 out of 104 applications. Our user studies show that 78% of end-users and 95% of developers agree with Keeper's detection and fixing results.

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

Additional Key Words and Phrases: software testing, machine learning, machine learning API

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 [2, 3, 35, 44, 71]. Indeed, many software applications in a variety of domains are incorporating ML APIs [22, 102]. Thus, effectively testing these applications—which this paper refers to as *ML software*—has become urgent.

To better understand this testing task, consider Phoenix [81], 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".

Authors' addresses: Chengcheng Wan, ccwan@sei.ecnu.edu.cn, Software Engineering Institute, East China Normal University, China; Shicheng Liu, sliu22@stanford.edu, Computer Science Department, Stanford University, United States; Sophie Xie, sxie2@berkeley.edu, Computer Science Division, University of California, Berkeley, United States; Yuhan Liu, yuhanl@uchicago.edu, Department of Computer Science, University of Chicago, United States; Henry Hoffmann, hankhoffmann@uchicago.edu, Department of Computer Science, University of Chicago, United States; Michael Maire, mmaire@uchicago.edu, Department of Computer Science, University of Chicago, United States; Shan Lu, shanlu@uchicago.edu, Microsoft Research and University of Chicago, United States.

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

Fig. 1. An example of using ML Cloud APIs [81].

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 accidental but inevitable mistakes of `label_detection`, which should be tolerated, or the application's incorrect use of the API, which should 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.

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

1) Infinite, yet sparse input spaces. The spaces of images, texts, or audio —typical input forms of cognitive ML APIs—are infinitely large, yet *realistic* inputs that are *relevant* to the software under test 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 [74], block replacement [109] , or mapping [65]) 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 relies on human judgment. 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 [48], a blood-vessel categorizer [50], an image region growth program [49], and a biomedical text processor [96]. Their design each targets a particular cognitive task and cannot be applied to general ML software.

3) Probabilistic incorrectness that is difficult to diagnose. When ML software produces outputs that differ from most human experts' judgment, 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—even a human expert makes mistakes in cognitive tasks occasionally. In other words, developers need to distinguish between 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, such as a crash, indisputably indicates a flaw in the software that must be fixed, whether it is within some APIs or within the software itself.

Note that, much recent work studies how to test [1, 4, 5, 10, 13, 24, 25, 28, 30, 31, 31, 32, 39, 43, 56, 63, 64, 66, 67, 74, 77, 78, 86, 91, 100, 110, 112, 114, 118, 119, 122], verify [26, 27, 27, 111], and fix [46, 58, 94, 97, 98, 115] 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 uses 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 judgment 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 experts , 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

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

consistent with common human judgment. In addition, Keeper also leverages anti-patterns from an empirical study [102] and conducts static analysis to detect code smells and provide fix suggestions.

Putting these all together, we have implemented Keeper which can be used either through a command-line script or a plug-in inside the VScode IDE [72]. 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 82 open-source Python applications that cover different problem domains and ML APIs. Keeper achieves 95% branch coverage on average for these applications. In total, Keeper covers 18–39% more branches than alternative techniques that directly use machine learning training datasets or random fuzzing. Keeper exposes 74 unique failures together with 19 code smells from 56 out of these 104 applications. We also conduct user studies with end-users and developers to evaluate Keeper and its testing reports. Around 78% of users agree with Keeper's judgment of accuracy failures and regard the software fixed by Keeper have better or similar quality. Around 95% of developers think Keeper's testing reports are helpful and its IDE plugin is easy to use.

Keeper's source code is publicly available at GitHub[2] and Zenodo[3].

2 BACKGROUND

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

2.1 ML Cloud APIs

In recent years, numerous cloud service providers [2, 35, 44, 71] have developed a range of machine learning cloud services that can be easily incorporated into software systems through cloud APIs. By using these APIs, sophisticated machine learning inferences can be conducted using pre-trained deep neural networks (DNNs) executed in the cloud.

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 [102].

Inputs. 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.

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

Fig. 2. An example of Google Cloud API with text input.

Outputs. The output of an 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

[2]https://github.com/mlapistudy/Keeper_artifact
[3]https://zenodo.org/records/10968650

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 [102].

2.2 ML software

Sometimes, ML APIs are only loosely connected with the remaining part of the software. Their output is directly presented to end users or stored without further use in the software, e.g., a translator. Testing this type of software simply needs to separately test ML APIs and the remaining part of the software. Therefore, they are 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. Take Figure 1 as an example, whether the software raises the alarm depends on the output of `label_detection` API. These cases present new challenges to software testing as discussed in Section 1 and hence are the focus of this paper.

2.3 ML Model Testing

ML models, typically in the format of neural networks, are widely studied on testing [1, 4, 5, 10, 13, 24, 25, 28, 30, 31, 31, 32, 39, 43, 56, 63, 64, 66, 67, 74, 77, 78, 86, 91, 100, 110, 112, 114, 118, 119, 122], verifying [26, 27, 27, 111], and fixing [46, 58, 94, 97, 98, 115], to improve their accuracy, fairness, and security. Another line of work studies implementation bugs of neural network architectures [47, 120] and other ML models [19, 99].

These works only focus on the cognitive task itself, aiming to improve the performance of ML models over a certain data set. On the contrary, this paper focuses on improving the software performance under its application scenario and aims to help developers integrate ML API into software. These works are orthogonal to the research problem of this paper.

3 OVERVIEW OF KEEPER

Keeper is a testing tool for software whose control flow is influenced by ML APIs. As shown in Figure 3, Keeper includes three major components: test input generation (Section 4), test output processing (Section 5), and code smell detection (Section 6).

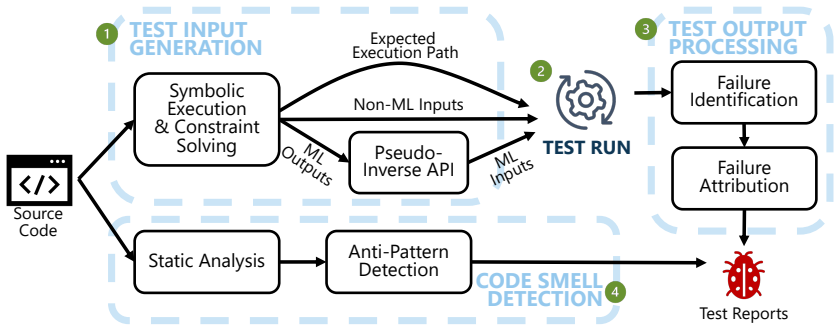

Fig. 3. An overview of Keeper.

Given the software source code and a function F to test[4], Keeper first generates test input with symbolic execution and constraint solving. Keeper represents all the function parameters of F as symbolic variables and generates symbolic path constraints for every branch of F and its callees. Keeper leverages pseudo inverse functions to solve the constraints of each feasible execution path and produces a diverse test suite that offers full branch coverage.

Once all the test inputs are generated, Keeper executes them and automatically judges output correctness based on whether the expected execution path is followed or not. Keeper then identifies and reports failures based on three symptoms: low accuracy, dead code, and fail-stops. After detecting a failure, Keeper attempts to patch the software by changing how ML APIs' output is used.

Besides defects that lead to incorrect software behavior, Keeper also detects code smells that degrade software performance. Motivated by an empirical study [102], Keeper conducts static analysis to detect performance-related anti-patterns in the scope of the tested function F and its callees. Once detected, Keeper provides fix suggestions to avoid unparalleled computation and skippable API invocations.

4 TEST INPUT GENERATION

In this section, we explain how Keeper handles cases when ML APIs are part of the path constraints and generate 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, BASIC-L [18], has 2440 million parameters. It takes in a 224×224 pixel image and generates the output after about 250 billion floating point operations. Solving a path constraint that involves this network with 224×224 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 4.1); and then synthesizes the ML-API inputs that are expected to produce those outputs using carefully designed pseudo-inverse functions (Section 4.2). As we will see, this decomposition not only avoids the complexity of directly applying symbolic execution to DNNs, but also helps judge the execution correctness (Section 5).

4.1 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 the 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 execute a certain path.

Keeper's input generation is built upon an existing symbolic execution engine, DSE [45]. The only tweak Keeper makes here is to have the symbolic execution engine sometimes generate 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 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

[4]Users of Keeper can specify which function to test, including the `main` function.

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

ML Task		Main Output	Pseudo-inverse Function
Vision	Image classification	image class	Search on internet, keyword: [image class]
	Object detection	object name	Search on internet, keyword: [object name]
	Face detection	face emotion	Search on internet, keyword: [emotion] + "human face"
	Text detection	extracted text	Print [extracted text] on an image
Language	Document classification	topic class	Search on internet, keyword: [topic class]
	Sentiment detection	score, magnitude	Select tweets from Sentiment140 dataset [33]
	Entity detection	entity name, type	Use text generation technique, seed: [name] or [type]
Speech	Speech recognition	transcript	Use speech synthesize technique on [transcript]

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.

Note that, while static analysis techniques could extract the keywords compared with ML API outputs, symbolic execution techniques are still essential to generate tests. *First*, keywords are not sufficient to infer the expected execution path, which is used for judging the correctness of a test run (see Section 5.1). While many applications use a whitelist to examine ML API output (i.e., `if API_output in whitelist`), a non-neglectable portion of them use a blacklist (i.e., `if API_output not in whitelist`) or other operations. *Second*, sometimes, multiple constraints are applied to the ML API output, which could only be resolved by symbolic execution. The nested if-structure in Figure 9 is an example. *Third*, symbolic execution is required to handle the pre-possessing step of ML API input and resolve test inputs unrelated to ML API (non-ML inputs).

4.2 Identifying ML API inputs

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

The pseudo-inverse function should have the following properties: *First*, it is not an analytical inversion of f. Ideally, it should be built independently from f (e.g., not based on the same training data set), so that it can help not only input generation but also failure identification in a way similar to N-version programming [7]. *Second*, it should be a semantic inverse of f, reversing the cognitive task performed by f in a way that is consistent with most human experts . 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*, it 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.

To achieve these goals, we design three types of pseudo-inverse functions as shown in Table 1.

4.2.1 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.

```
def smart_can(img):
  labels = client.label_detection(image=img)
  classes = [x.desc for x in labels]
  for c in classes:
    if  c == "food":
      return "organic"
    if c == "paper" or c == "aluminum":
      return "recyclable"
  return "non-recyclable
```

Fig. 4. A smart can application, Heap-Sort-Cypher [40]

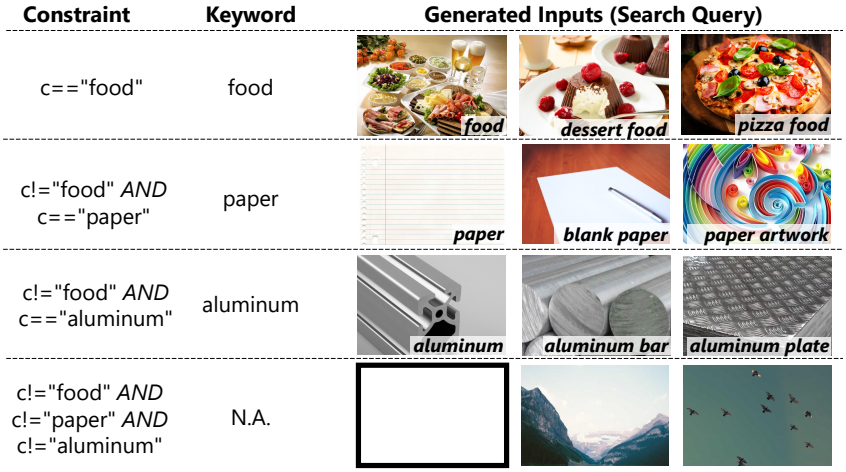

Fig. 5. Keeper-generated test cases for Figure 4

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 [15]. Their top search results typically match the common human judgment. Second, they are not an analytical inversion of ML APIs, and we will use engines from different service providers 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 [12] image search engine and uses the desired label description or object name as the search keyword. The face-detection API detects human faces together with emotions (e.g., "joy", "sorrow", etc.) in an image. 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 topics based on its content, like "art", "health", and "sports". Keeper uses the desired topic name as a keyword and searches it at (1) knowledge graph websites, e.g., Wikipedia [107]; and (2) Bing web search engines. Keeper then uses the text extracted out from each returned web page as the ML API input.

Search engines are good, but not good enough. As search engines typically rank results by their relevance and quality, it is likely that the top-ranked images/text share similar styles or belong to a

particular subcategory of the keyword, which is not ideal for software testing. For example, the top-ranked "paper" images from Bing are mostly notebook pages, while the software input might be colored paper images.

To further improve the variability of generated tests, Keeper utilizes the phrase association feature of search engines to find subcategories of the given keyword. For example, the keyword "paper" is associated with the phrase "blank paper", "paper artwork", and others. Keeper then uses the keyword and its m subcategories to query the search engine and obtain their top results, where $m = \lfloor\sqrt{|I|}\rfloor - 1$. The images in each row of Figure 5 are the top search results returned by Bing and the actual search queries are shown at the bottom of the images. The only exception is the last row: when there is no specific keyword requirement, Keeper uses a blank image and images from a random-image generator [69] for testing.

Note that, some cloud services use both a label name (e.g., "cat") and the corresponding unique id (e.g. "/m/01yrx") to describe its categorical result. To support keywords in either format, Keeper maps the unique id to the label name using official documents.

4.2.2 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 both printed and hand-writing fonts; different font settings produce different test images. Keeper uses a blank image and some random images as the background. Figure 6 shows some of the test images that Keeper generates for application wanderStub [104], which has a branch checking if the input image contains "Total".

Fig. 6. Test inputs generated for wanderStub [104].

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 generation model [84] 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 [11] 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.

4.2.3 ML benchmarks for pseudo inversion. Reversing **sentiment detection** API faces two challenges. First, it does not directly output a categorical result of the prevailing emotional opinion within the text. Instead, it returns two floating-point numbers for developers to derive emotion categories from. There is no perceivable way to generate text that can offer the exact numbers. Second, even if we just hope to generate text that contains positive or negative emotions, no search engine can accomplish this.

Facing these challenges, Keeper resorts to the Sentiment140 dataset [33], which contains 1,600,000 tweets, manually labeled 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 [23] 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 [55]. 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 ML APIs, which makes the test inputs biased towards the ones APIs can perform well on and hence less likely to reveal problems. Finally, we do not use generative models to synthesize new data, as this approach requires much training data and ends up generating non-real-world data that has a similar distribution with the training set, whose limitations we discussed earlier.

4.2.4 Solving corner cases. Keeper also tweaks three types of pseudo-inverse functions to fit some special API usage scenarios.

Path without constraints. If the execution path does not have any constraint on ML API output, Keeper uses random keywords from the API category list and a pre-defined set.

Empty set output. If the desired ML API output is an empty set, Keeper uses blank images, empty string, and muted audio for its input.

APIs' sharing input. If multiple *vision* APIs share the same input image, Keeper concatenates the keywords of all APIs, except for text-detection API, and invokes pseudo inverse functions to collect real-world data. Keeper then uses these images as background and prints text strings desired by text-detection API. If multiple *language* APIs share the same input, Keeper concatenates text strings obtained from each pseudo-inverse function. If multiple *speech* APIs share the same input, Keeper concatenates the desired transcript and invokes the pseudo inverse function.

4.3 Input Generation Summary

Overall, Keeper generates test inputs for any function F in the following steps.

First, for each path whose constraints are satisfiable, Keeper's symbolic execution (Section 4.1) generates a set of inputs $\mathbb{I}$. Keeper first resolves non-ML inputs with standard symbolic execution. 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 on o to generate a set of image/text/audio inputs for f (Section 4.2). 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 applies symbolic execution on that pre-processing code, with additional constraints on f's input, 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. Keeper's ability to handle pre-processing is limited by the symbolic execution engine, which currently only supports text.

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 clip i generated by Keeper, Keeper changes the file path embedded in the software to a path that points to i.

5 TEST OUTPUT PROCESSING

5.1 Failure identification

Keeper looks for three types of failure symptoms: low accuracy, dead code, and generic failures.

5.1.1 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 judgment; (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 judgment; 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 produce outputs that are inconsistent with common human judgment.

Specifically, for the input set $\mathbb{I}_c$ that is generated to cover a branch b and satisfy one of its sub-condition c, Keeper checks which of them exercises b at run time, no matter which sub-condition of b is satisfied, denoted as $\mathbb{I}_c^{\text{succ}}$ and calculates the *recall* of b (i.e., $\frac{|\mathbb{I}_c^{\text{succ}}|}{|\mathbb{I}_c|}$). If the recall of any input set generated to cover b 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 4.2.3. 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.

Root causes of accuracy failures. These accuracy failures are **not** equivalent to low precision or low recall of the ML API itself. While the latter is a possible root cause of the former, incorrect software implementation is also likely ti lead to accuracy failure. Take the fire alarm application in

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

Fig. 7. A fix of Figure 1 suggested by Keeper.

```
topics = language_client.classify_text(document)
for topic in topics:
  if topic.name=="Sensitive Subjects" or "Adult":
    record_profanity()
```

Fig. 8. Dead-code bugs in **Soap** [93]

Figure 1 as an example, due to improper branch conditions, the software wrongly silences the alarm when the ML API correctly recognizes flame from a forest fire image. In addition, an inaccurate ML API might be tolerated by software, either due to careful software implementation, or insensitive application scenarios. Therefore, 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 *improper label* problem. For example, the `label_detection` API does not return "fire" as a top-3 label for many fire images, which 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 label that does not exist in the API's label set and can never be the output. For example, an application [40] compares the `label_detection` output with a non-existing label "clothes", while "clothing" is the valid label.

5.1.2 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 an ML API output with a non-existing label. For example, a weather application [85] checks the output of `label_detection` against the string "gloves". Unfortunately, among the 20,000 category labels that could be output by this API, none of them is "gloves". Instead, "glove" 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 **Soap** [93]. Soap uses `classify_text` to judge whether an article contains sensitive subjects or adult content. Unfortunately, it wrongly uses `"Adult"` instead of `topic.name == "Adult"` in its branch condition, making the if-statement always True. It will regard every article identified with at least one topic as containing adult content!

5.1.3 Generic failures. These have symptoms like crashes that do not require special techniques to observe. Compared with traditional testing techniques, Keeper offers extra benefits 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 an empty set (Figure 9).

5.2 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 consider using a different, more accurate ML API, or adding extra input screening or pre-processing.

```
text = client.text_detection(image=img)[0].description
if "take" in text or "takes" in text:
    if "meals" in text:
        m = re.search("(\w+)\s+meals'", text)
        meals = m.group(0)
if "mg" in text:
    m = re.search("(\S+)\s+mg", text)
    dosage = m.group(0)
```

Fig. 9. Generic failure in **Pill-Identifier** [76]

5.2.1 Label changes for categorical outputs. When branch b compares an ML API categorical output with a set of labels, Keeper tries to find a minimum extension of such label set with three goals: (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.

Vision tasks. Keeper generates patch at branch level, considering the impacts of all the labels compared with `o`. To fulfill 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 [106]. For example, "plant" 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 $\texttt{l} \in L$ 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). If multiple labels have the same improvement, Keeper selects the one which has more neighbors in the set. 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.

Language tasks. Similar to vision tasks, Keeper uses the same greedy algorithm to expand the compared labels from L. As the categories of document classification tasks are virtual concepts that are not well described in the knowledge graph, Keeper adopts class hierarchy to capture the semantic relationship between these virtual concepts. Keeper filters out every label in L that is neither a super/sub-class of nor sharing a common super class with `label0`.

Note that, utilizing knowledge graph and greedy algorithm, Keeper naturally suggests the valid synonym for a non-existing label compared with o. To serve the non-expert developers, we avoid using technical terminologies to describe the patch and its impact, but use plain languages instead. 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 the software will behave correctly on 45% more test cases generated by Keeper.

5.2.2 Threshold changes for numerical outputs. When branch b checks the numerical outputs of the sentiment-detection API, Keeper tries to further subdivide b to better differentiate text inputs with different emotions. Keeper checks all the inputs that exercise b and applies logistic regression

```
  response = client.analyze_sentiment(text)
  score, mag = response.score, response.magnitude
  if score > 0.5:
    feelings = "really like"
  elif score > 0:
-   feelings = "like"
+   if 3.0*score - 0.3*mag >= 0:
+     feelings = "like"
+   else:
+     feelings = "neutral"
  elif score > -0.5:
    feelings = "dislike"
  else:
    feelings = "really dislike"
```

Fig. 10. A fix of **Animal-Analysis**[6] suggested by Keeper

with the {score, magnitude} output as feature vectors and the labeled emotion as a class. If the prediction accuracy is larger than α, Keeper then suggests developers create a new branch under b, using the linear formula of the logistic regression model as the branch checking threshold, as shown in Figure 10. Keeper also informs developers how many text inputs are correctly differentiated with this patch.

6 CODE SMELL DETECTION

While Keeper-generated tests are effective to reveal semantic bugs that lead to software misbehaviors, they cannot detect the performance bugs that slow down the software execution. Motivated by the anti-patterns identified by an empirical study of ML software [102], Keeper uses static analysis to detect performance-related code smells and provide patch suggestions.

Given a function F to test, Keeper first conducts function call graph analysis to trace all the functions invoked during the execution of F. Keeper then in-lines them to get an in-lined function F'. When there exist recursive calls, Keeper only in-lines each of them once. Keeper detects code smells within the scope of F', through pattern-based static analysis. Unlike existing performance optimization techniques that focus on general software or compilers, Keeper incorporates prior knowledge of ML Cloud API to detect performance bugs that are uniquely severe in ML software. Particularly, Keeper focuses on three types of code smells.

Detect synchronous usage of asynchronous API. Unlike classical API design, asynchronous ML APIs typically are much slower than the corresponding synchronous version, as they are designed for large-scale time-insensitive tasks. Therefore, invoking asynchronous APIs in a blocking way would greatly slow down software execution. To detect such code smell, Keeper examines whether F' invokes an asynchronous ML API without using any multi-thread/process library. The multi-thread/process library is identified by comparing the library-import and library-function-invocation statements with a configurable list[5] that includes libraries that support parallel computing maintained by Keeper. Once detected, Keeper suggests developers either invoke ML API in another thread to improve parallelism or replace it with synchronous API to reduce latency.

[5]The default list is obtained through Python official guide of parallel computing: https://wiki.python.org/moin/ParallelProcessing

Detect unparalleled API invocations. ML API invocations consume limited local computing resources, as the inference procedure is managed by the cloud. Invoking multiple ML APIs at the same time won't lead to resource contention, but greatly improves software performance. Many cloud ML services offer interfaces to invoke ML APIs in parallel, e.g., `annotate_image` and `annotate_text` API from Google AI. For any application that invokes multiple ML APIs, Keeper conducts data flow analysis on ML API inputs. If two ML APIs share the same input, Keeper examines whether they are invoked with the parallelism interface or other multi-thread/process libraries. If not, Keeper reports the code smell and suggests developers use the parallelism interface to parallel ML API invocations.

Detect repeated API invocations on constant inputs. Depending on the type of ML task and network bandwidth, an ML API invocation typically takes from 0.2 to several seconds. Therefore, invoking ML API on constant inputs would lead to a non-negligible performance drop. To detect such code smell, Keeper conducts data dependency analysis for the ML API input. Keeper focuses on the ML API that is invoked inside loops, or its caller function is invoked in a loop or an event handler. Keeper reports a code smell if the ML API input is a constant variable unrelated to the file path. Keeper then suggests the developer "cache the API result or move the API call outside the loop".

7 IMPLEMENTATION

7.1 Core algorithm

We have implemented Keeper for Python applications that use Google Cloud AI APIs [35], the most popular cloud AI services on Github [102]. The core algorithm of Keeper is general to other languages and ML Cloud APIs. Keeper uses DSE symbolic execution engine PyExZ3 [45] and constraint solver CVC [9] . Keeper uses Python built-in trace back tool to check branch coverage, and Pyan [68], AST [83] and Jedi [38] for static analysis analysis. Keeper uses scikit-learn library [89] for linear regression models and pillow library [20] for image processing.

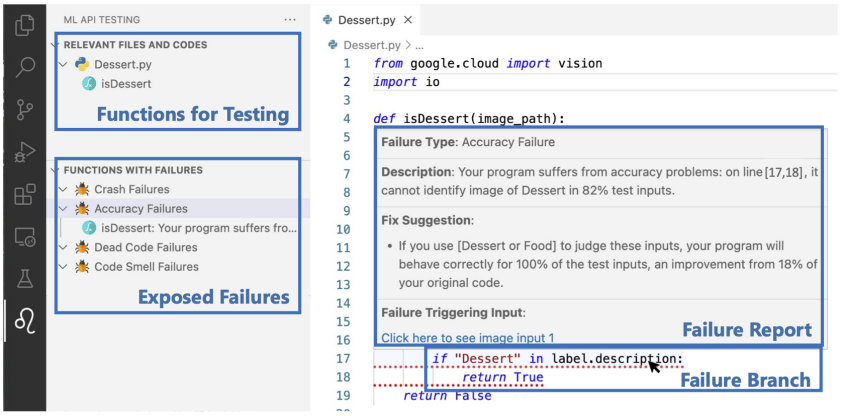

Fig. 11. Keeper IDE plugin interface

7.2 IDE plugin

As illustrated in Figure 11, we have implemented an IDE plugin for visualized interaction with Keeper, as the debugging and fixing of accuracy failures and performance bugs particularly require developers' participation. In addition, the timely feedback helps programmers preventing defects in early development stage [88, 116, 117]. The plugin is an extension in Visual Studio Code [72], a

Table 2. Statistics of our benchmark suite (*: The ML task is associated with multiple ML APIs)

	# Apps	ML Task (# Apps)
Vision	49	Image classification* (26), object detection (19), face detection (4), text detection* (12)
Language	42	Document classification (16), sentiment detection (22), entity detection* (14)
Speech	13	Speech recognition* (13), Speech Synthesize (1)

popular code editor supporting multiple languages. For any Python software, Keeper first identifies all functions that invoke ML APIs directly or indirectly through callees, and displays them on the sidebar, under "RELEVANT FILES AND CODES" in Figure 11. From that list, developers can select the function to test and provide type information of function parameters, as Python is a dynamically typed language. Keeper will then start the testing, which usually takes 1–2 minutes, and show logs of testing progress. At the end of the testing, all exposed failures are grouped by categories and listed in the sidebar, right under "FUNCTIONS WITH FAILURES" in the figure. Source code related to each failure is highlighted with underlines, together with a hovering window that offers detailed information including failure type, description, patch suggestions and failure triggering inputs.

Keeper designs bug reports to be easy to understand and follow. Guided by a small-scale user study with expert and non-expert developers, Keeper improves its text guidelines by using the least technical terminologies and avoiding ambiguous phrases. It also provides fault localization guidance when all patch attempts fail. Besides the patch suggestions introduced in Section 5 and 6, Keeper also suggests "Please validate the text/image/audio input of ML API before invoking it." and "The source code probably contains syntax errors." when software crashes on some or all of the test inputs. If the software suffers a dead-code failure, Keeper suggests developer examine branch predicates for a syntax error.

8 EVALUATION

Our evaluation aims to answer several questions:

RQ1. (Sufficiency) Does Keeper help improve the branch coverage in testing?
RQ2. (Effectiveness) Is Keeper able to find bugs and suggest fixes during its testing?
RQ3. (End-user experience) How do software users feel about the accuracy failures?
RQ4. (Developer experience) How do software developer feel about Keeper?

8.1 Methodology

8.1.1 Applications. As shown in Figure 2, we evaluate Keeper using 104 Python applications that are from two sources. 1) From the 360 open-source applications assembled by a previous study of ML APIs [102], we find 45 Python applications that use ML APIs in a non-trivial way (i.e., the API output affects control flow). 2) From the GitHub code search engine, we additionally check the first 800 Python applications that use ML APIs and found 59 applications that use ML APIs in a non-trivial way. These 104 applications use a range of ML APIs, including vision (49 apps), language (42 apps), and speech (13 apps). Their sizes range from hundreds to millions of lines of code, with 185,235 lines being the average size.

For more than half of the applications (55), 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. During testing, we evaluate the branch coverage and exposed failures of these functions-to-test and their callees. The average number of branches is 14.

8.1.2 Baselines. We compare Keeper with 3 alternative techniques, with each technique generating 100 test inputs for each function under test.

Table 3. Average branch coverage across 104 applications.

	Vision App.	Language App.	Speech App.
Keeper	93.1%	96.8%	93.1%
Keeper*	92.2%	96.0%	93.1%
Random-Real	75.2%	84.3%	60.3%
Random-Real-Noise	74.2%	75.8%	60.3%
Fuzzing	49.3%	68.8%	35.8%

(1) *Random Real*: it randomly picks inputs from well-established data sets, including ImageNet [23] that contains 14 million images, Twitter US Airline Sentiment [51] that contains 15,000 tweets, and LibriTTS dataset [113] that contains 585 hours of English speech audio.

(2) *Random Real + Noise*: it adds random noise to inputs picked by *Random Real*. For an image, it randomly adds noises following Gaussian distribution; for a text input, it randomly changes the word orders. It does not add noise to audio, as we find 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*: it uses a coverage-based fuzzing tool pythonfuzz [29] to generate tests. For every image input, it uses an integer list to fill its RGB matrix in a repeated way. For every text input, it generates ASCII character sequences. For audio inputs, it directly generates the audio data.

This paper is an extension of our earlier work [103], supporting code smell detection, and improving the variety of generated tests, coherence of patch suggestions, and readability of the failure report. To conduct an ablation study, we also compare Keeper with its original version Keeper*.

8.1.3 User studies. Unlike traditional software failures like crashes, accuracy failures could be subjective to human judgment. Therefore, we conduct two user studies, one with 100 participants through an online survey and Amazon Mechanical Turk, and one with 21 Python developers through Zoom interviews and Prolific [82], to see whether end-users and developers agree with Keeper's judgment and think about Keeper. We will present the details in Section 8.3.

8.2 Software testing evaluation

8.2.1 Branch coverage. For each of the 104 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 summarizes the results. The statistics and detailed experimental results of each benchmark application are available in Appendix **??**.

Across different types of applications, Keeper consistently achieves high branch coverage, around 95% 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.

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 18% and 33% 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 69% branch coverage. This is probably because language APIs' output, like document class or entity type, has much less variation than that of vision and speech APIs.

Table 4. Unique failures exposed by Keeper.

Failure	Root Cause	Related ML Task	Keeper	Keeper*	RReal	RReal+N	Fuzz.
Generic	Out-of-bound accesses	Text detection, entity detection	7	7	6	6	4
	Missing input validation	Document classification	5	5	-	-	-
	Missing type conversion	-	1	1	1	1	1
Accuracy	Improper labels	Image classi., object detect., document classi.	24	19	-	-	-
	API limitations	Image classi., object detect., document classi.	18	15	-	-	-
	Improper threshold	Sentiment detection	9	9	-	-	-
Dead-code	Typos	Image classi., document classi., text detect.	4	-	-	-	
	Non-existing label	Image classification	6	6	-	-	-

Answer to RQ1 (Sufficiency): Keeper achieves 95% average branch coverage across all types of ML applications, much higher than the alternative solutions.

8.2.2 Failure exposing and attribution. As shown in Table 4, Keeper exposes many failures by running those 100 test inputs it generated: 74 failures from the latest version of 45 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. In comparison, alternative testing techniques miss 6–9 generic failures caught by Keeper. Furthermore, unlike Keeper, they cannot automatically recognize accuracy failures and dead-code failures.

Accuracy failures. Among the 51 accuracy failures exposed by Keeper, 9 are related to threshold checking for the sentiment detection API. For these failures, Keeper manages to suggest better checking threshold to differentiate positive text from negative ones.

There are 42 accuracy failures related to label checking for the APIs with categorical outputs. Keeper manages to fix 24 of them by expanding the corresponding branch with 1–3 additional labels. The failure in Figure 1 is one such example. As another example, one application [93] checks if the output of `classify_text` contains "social issues & advocacy". This branch has a low recall of 43%. Keeper suggests adding the superclass "people & society" to the label set, which would improve the recall to be above 75%. This also reflects a special nature of categorical APIs: while they make mistakes on subclasses from time to time, their judgment of superclasses is usually reliable.

For the remaining 18 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. For example, a clothes checker application [21] looks for the non-existing label "person" from the image-classification API. Keeper suggests checking the synonym "human" instead, which increases the branch's recall to 45%, but still below 75%. Actually, due to the limitation caused by its design, this API tends to describe human images by clothes and hairstyle. However, the branch's recall would be improved to 100%, if the application uses the object-detection API instead.

Dead-Code failures occur in 10 applications. Six of them are due to non-existing labels. Four of them are because of typos in branches that process ML API output, like the one in Figure 8.

Generic failures are mainly caused by out-of-bound accesses to lists returned by ML APIs and missing input validations for ML APIs, as shown in Figure 9. One crash is caused by a 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 three false positives in total (they are not included in Table 4). One application tries to detect sensitive documents 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 relevant test inputs, and hence incorrectly reports a dead-code failure. The other two applications have a branch that gets covered only when an ML API generates a

```
redundant_stuff =['Food','Fruit','Plant','Vegetable',...,'Drink','Dessert','Cup']
temp_entry = []
labels = client.label_detection(image=img)
objects = client.object_localization(image=img)
for label in labels:
    if label.description not in redundant_stuff:
        temp_entry.append(label.description)
for object_ in objects:
    if object_.name not in redundant_stuff:
        temp_entry.append(object_.name)
if 'Packaged goods' in temp_entry:
    text = client.text_detection(image)
    process(text)
```

Fig. 12. Unparalleled API invocations in Omakase [75].

```
request = client.long_running_recognize(audio)  # async speech-recognition API
result = request.result(timeout=90)             # wait until request finished
images = process_transcript_and_download_images(result)
video_frames = initialize_video(audio.duration)
video = fill_in_images(video_frames, images)
```

Fig. 13. Synchronous usage of asynchronous API in SIVG [92].

specific output with low confidence. Keeper is not effective at generating low-confidence inputs and wrongly reports an accuracy failure.

Threshold setting. As discussed in Section 5.1, 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 create 12 more failure reports; decreasing α to 0.6 would have 5 fewer failure reports.

8.2.3 Code smell detection. Keeper detects code smells in 19 applications without false alarms.

Sixteen of them sequentially invoke multiple vision or language APIs on the same input. After adopting the patch suggested by Keeper, they achieve 1-2x speed up. In a news summary application [60], the developer uses the sentiment-detection API and entity-detection API to analyze news articles and highlight the topic sentences. Although sharing the same input, these two APIs are invoked separately, which almost doubles the execution time. As another example in Figure 12, Omakase [75] sequentially invokes both image-classification and object-detection API to identify all the food in a picture. If there exists any packaged goods, it invokes text-detection API to obtain its content. Through Keeper-generated tests, Keeper reports that Omakase fails around 5% of drink images, which is tolerable. However, paralleling these ML APIs could offer 2X or even 3X speedup, which is a non-neglectable performance improvement.

Two applications use asynchronous APIs in a blocking way. For example, **SIVG** [92] in Figure 13 uses an asynchronous speech-recognition API to analyze the audio input, download images based on the transcript, and use the transcripts to generate video frames. Although video initialization does not have data dependency with the ML API invocation, the developer executes these two time-consuming operations sequentially and thus degrades the performance. Actually, the software is completely idle when waiting for the API response (line 2), which provides an ideal chance for parallelism.

```
1 while True:
2     text = speech_client.recognize(audio).result
3     if (text == "hey lisa") or (text == "hi lisa") or (text == "lisa"):
4         call_lisa = True # waked up
5         audio = text_to_speech_client.synthesize_speech("Hi, how can I help you?")
6         play_audio(audio)
7     elif call_lisa:
8          process_lisa_command(text)
```

Fig. 14. Repeated API invocations in Lisa Assistant [62],

Keeper detects repeated API invocations in a voice assistant application [62]. As shown in Figure 14, this application listens to the microphone, uses speech-recognizes API to transcribe users' voice commands, and generates audio replies through speech-synthesize API. While passing all the generated tests, it actually has a severe defect that hurts performance: the speech-synthesize API in line 5 will be invoked with the same input string every time the user awakes the voice assistant. If this application caches the generated audio clip and replaces the future API invocations with it, around 1 second could be saved for each waking phase.

8.2.4 Ablation Study. Compared with Keeper*, Keeper improves around 1% average branch coverage for vision and language applications. By improving test input variety through phrase association mechanism, Keeper-generated tests are more likely to cover the application scenario, especially for those "long-tail" applications. Take a property management application [87] as an example, it uses the keyword "wall" to examine the image-classification results of an indoor image. However, using this keyword, the image search engine only returns brick walls, whose corresponding ML API output is typically about bricks. Instead, using the associated phrase "office wall", Keeper successfully finds indoor images with wall and fulfills the branch condition.

Besides improving branch coverage to increase the chance of exposing more failures, we believe test variety improvement is also useful to understand the limitations of ML API and to guide developers in fixing their implementation. Particularly, Keeper detects 8 more accuracy failures than Keeper*, and is able to fix 5 of them. These failures are wrongly reported as dead-code failures by Keeper*, and also failed to be patched.

Keeper also detects 19 more performance-related software defects than Keeper*, which proves the importance of Keeper's code smell detection mechanism.

Answer to RQ2 (Effectiveness): Keeper reveals 71 failures together with 19 code smells from 56 out of the 104 tested applications. The symptoms cover low accuracy, crashes, dead-code, and performance loss. The patches suggested by Keeper successfully fix 29 failures and 16 code smells. In comparison, alternative solutions only detect 5 to 7 generic failures.

8.3 User studies

8.3.1 Study with end-users. To better evaluate the accuracy failures and the code changes suggested by Keeper, we recruit 100 participants on Amazon Mechanical Turk 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 15);

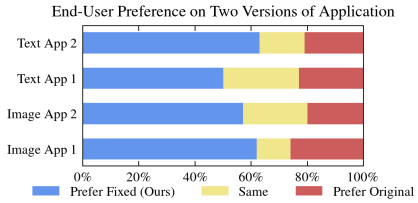

Fig. 15. End-user preference of applications.

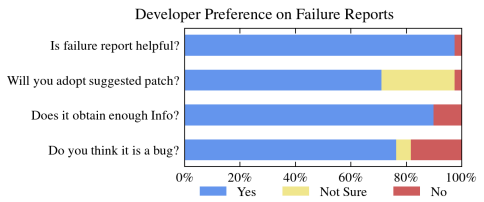

Fig. 16. Developer preference of failure reports.

Table 5. System usability scale of Keeper. The score of positive (negative) questions ranges from 1 to 5 for strong disagree (agree) to strong agree (disagree)

Question	Average Score
I think that I would like to use this tool frequently.	4.33
I needed to learn a lot of things before I could get going with this tool.	2.91
I found this tool unnecessarily complex.	3.52
I thought this tool was easy to use.	4.47
I think that I would need the support of a technical person to be able to use this tool.	3.24
I found the various functions in this tool were well integrated.	4.47
I thought there was too much inconsistency in this tool.	3.23
I would imagine that most people would learn to use this tool very quickly.	4.33
I found this tool very cumbersome to use.	3.22
I felt very confident using this tool.	4.09

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 are compensated $5 after the survey.

A summary of the user study results is shown in Figure 15. As we can see, in all cases, a dominant portion of end-users prefer the version with changes suggested by Keeper over the original version, supporting Keeper's judgment about accuracy failures and Keeper's attempt to improve software quality. At the same time, we also notice 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 experts often do not agree with each other on these tasks.

Answer to RQ3 (End-user experience): Most end-users think that the Keeper-fixed software has better quality than the original one, supporting Keeper's failure detection and fixing capability.

8.3.2 Study on developers. To further evaluate the usability of Keeper and developers' attitude towards Keeper-generated tests and patches, we conduct an interview-based user study.

We recruit 14 male and 7 female participants who have Python programming experience through a widely-used user study platform Prolific [82]. The participants are between 18 and 60 years old. Six of them are between the ages of 18-24 and nine are between 25-34. In terms of educational background, 3 participants have a Master's or Ph.D. degree, 11 have a Bachelor's degree, and the remaining 6 have a lower degree. Among all 21 participants, 18 have computer science background and 17 have knowledge of machine learning.

Given ML API official documents and manuals, participants are asked to implement two functions, one for image analysis and one for text analysis, with the code skeleton provided by us. They are then shown Keeper instructions and asked to test their implementations independently. For any

```
1  THRESHOLD = 0.5
2  def is_about_food(article_text):
3      response = client.classify_text(article_text)
4      if response is None:
5          return False
6      for category in response:
7          name = category.name.lower()
8          if category.confidence > THRESHOLD and ("nutrition" in name or "food" in
       name or "cuisine" in name):
9              return True
10     return False
```

Fig. 17. A code snippet written by Participant-1, judging whether an article is about food.

failures reported by Keeper, participants are asked about whether they think the failure indeed reflects a software bug; how they would fix the code; whether they think Keeper is helpful; and others. Finally, the participants are asked to fill out a System Usability Scale (SUS) questionnaire [57]. To reduce bias, the participants are told that the goal of this study is to understand how people implement ML software, and the testing tool is part of the development process. This whole process is conducted through Zoom, with one researcher remotely interacting with the participant. The whole session took 60–90 minutes. Participants are informed of a fixed compensation of $20 before the interview. At the end of all interviews, two researchers code the interview data independently and then meet to resolve disagreements.

In total, Keeper reports 32 accuracy failures and 7 dead-code failures for the 42 implemented functions. The participants each has 1–2 failures exposed by Keeper from their code. Keeper suggests code changes for all 32 accuracy failures and points out the possibly problematic code snippet for 6 of the dead-code failures.

Figure 16 summarizes developer preference of failure reports. As we can see, almost all failure reports (38 out of 39) are regarded as helpful. For most of the failure reports (27 out of 39), participants say they would definitely adopt the patch suggested by Keeper. Participants are not sure about the suggested patch in 10 reports, because they want to inspect Keeper-generated test cases before making final decisions. Note that, most participants (19 out of 20) agree with Keeper-generated test cases. The other participant feels that some tests are harder than his/her expectation and the software failure is totally acceptable. This further illustrates the subjectivity when judging software output correctness.

Participants strongly agree that most failure reports (29 out of 39) pointed out bugs in their programs. There are 7 reports, all about accuracy failures, where participants feel the failures do not mean their code is buggy, although in all cases they are willing to adopt the code changes suggested by Keeper. One participant described that “It is not a bug, but could be improved.” (P10). Some participants complained about ML API, saying that “I hope there is a better ML API design.” (P1); “Is there a way to customize the categories of ML API? I want it to be more specific.” (P9). We believe this reflects the subjective nature of cognitive tasks.

Figure 17 shows the code snippet written by Participant-1, aiming to judge whether an article is talking about food in a forum application. After a few hand-coded unit tests, Participant 1 finally uses keywords “nutrition”, “food”, and “cuisine” to examine the result of `classify_text` API. However, it only passes 71% of Keeper-generated tests to cover Line-9. Keeper suggests additionally checks keywords “agriculture” and “forestry” to improve branch recall to 78%. Going through the

failure report, Participant 1 accepts the patch, saying "Given the flaw of ML API, this patch could improve software."

The results of the SUS questionnaire are summarized in Table 5. Overall, the participants gave positive responses to most of the questions, indicating that Keeper is easy to use. Only the second question on learning cost gets an average score of 2.9 (neither agree nor disagree), as Keeper requires background knowledge of ML APIs. The participants tell us that "I'm definitely interested to use this tool when I'm developing such ML software. It is fun to use it." (P7); "The tool is easy to use. Just need to press the buttons" (P15); "Making testing automated is great." (P21).

Answer to RQ4 (Developer experience): A dominant portion of Keeper's failure reports are agreed by the developers and the corresponding patches are accepted. Most developers find these failure reports informative and helpful. They also think Keeper is user-friendly and easy to use, admiring Keeper's algorithm and interface design.

8.4 Threats to Validity

Internal threats to validity. Keeper assumes that the top results of search engines are mostly consistent with human judgment, 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 recall sometimes comes with a decrease in 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. Exactly considering all these situations, we implemented Keeper IDE plug-in, aiming to help developers make informed decisions 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. The participants of our user study may not represent all real-world end-users and developers. The applications in our benchmark suite may not represent all real-world applications. Our tool is only tested with Python applications using Google AI, not other ML Cloud API services. Keeper only covers cloud APIs with pre-trained DNNs designed for general purpose use, not user-defined DNNs based on their specific needs.

9 RELATED WORK

9.1 ML-related software

Prior work studied development phases [3, 42, 53, 54, 121] of software that contains machine learning components. They do not look at how to test such software.

Other work focuses on selecting ML APIs [17] or ensemble their results [16, 108] from different cloud service providers, to improve the accuracy of recognition tasks. They do not consider software context and also cannot judge the correctness of each individual input. Another work [102] manually identified anti-patterns from software that uses ML APIs. Keeper differs from this study by proposing testing techniques that can *automatically* expose failures and attribute failure causes. Keeper also proposes an automated code smell detection module based on these anti-patterns. Furthermore, due to the different design goals, Keeper also covers more failure root causes than the previous

study. In the 45 applications that are evaluated both by Keeper and the previous study, Keeper automatically exposed 35 failures, among which only 6 were also identified by the previous study.

Another line of work [41, 52, 59, 61] 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.

9.2 ML-related testing

Much research has been done for testing [1, 4, 5, 10, 13, 24, 25, 28, 30, 31, 31, 32, 39, 43, 56, 63, 64, 66, 67, 74, 77, 78, 86, 91, 100, 110, 112, 114, 118, 119, 122], verifying [26, 27, 27, 111], and fixing [46, 58, 94, 97, 98, 115] neural networks, in terms of accuracy, fairness, and security. Other work studies implementation bugs of neural network architectures [47, 120] and other ML models [19, 99]. They are orthogonal to Keeper.

As discussed in Section 1, some previous work looked at how to test specific software that contains ML components [48–50, 96]. Unfortunately, their solutions do not apply to general ML software. For example, one work trained an 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 [48]. To test a blood-vessel image categorizer, previous work [50] generates blood-vessel images with certain density, branches, and other features, and uses these features to generate output ground truth. Previous work [49, 96] uses metamorphic approaches to test entity detection and image region growth programs. They require application-specific rules about the inputs and outputs relationship (e.g., after we concatenate inputs of entity detection, the output becomes the concatenation of individual outputs [96]).

Prior work studies automatic testing and bug detection of machine learning APIs, including frameworks for implementing neural networks [8, 14, 37, 73, 80, 95, 101, 105] and REST APIs that provide machine learning solutions [34, 36, 79]. They focus on the implementation inside ML APIs, not how they interact with other software components.

9.3 Test generation using search engines

Previous work [70, 90] 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.

10 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 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. It also conducts static analysis to detect code smells. We evaluate Keeper with a variety of real-world machine learning applications and achieve high code coverage with a small set of test cases. It identifies bugs that lead to software crashes, lower inference accuracy, dead code, or performance loss. Our user studies show that most end-users and developers agree with Keeper's judgments and find it helpful.

ACKNOWLEDGEMENT

We thank the reviewers for their insightful feedback. The authors' research is supported by NSF (CNS1764039, CNS1956180, CCF2119184, CNS1952050, CCF1823032, CNS2313190), ARO (W911NF 1920321), and a DOE Early Career Award (grant DESC0014195 0003). Additional support comes from the CERES Center for Unstoppable Computing, UChicago Marian and Stuart Rice Research Award, Microsoft research dissertation grant, and research gifts from Facebook.

REFERENCES

[1] Raja Ben Abdessalem, Shiva Nejati, Lionel C Briand, and Thomas Stifter. 2018. Testing vision-based control systems using learnable evolutionary algorithms. In *ICSE*.
[2] Amazon. 2022. Amazon artificial intelligence service. Online document https://aws.amazon.com/machine-learning/ai-services.
[3] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. Software engineering for machine learning: A case study. In *ICSE-SEIP*. IEEE, 291–300.
[4] Saleema Amershi, Max Chickering, Steven M Drucker, Bongshin Lee, Patrice Simard, and Jina Suh. 2015. Modeltracker: Redesigning performance analysis tools for machine learning. In *CHI*.
[5] Rico Angell, Brittany Johnson, Yuriy Brun, and Alexandra Meliou. 2018. Themis: Automatically testing software for discrimination. In *ESEC/FSE*.
[6] Animal-Analysis. [n. d.]. An image sharing platform. Online document https://github.com/OkapalDominic/animal_analysis.
[7] Algirdas Avizienis. 1995. The methodology of n-version programming. *Software fault tolerance* 3 (1995), 23–46.
[8] Soheil Bahrampour, Naveen Ramakrishnan, Lukas Schott, and Mohak Shah. 2015. Comparative study of deep learning software frameworks. *arXiv preprint arXiv:1511.06435* (2015).
[9] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Computer Aided Verification*. Springer, 171–177.
[10] David Berend, Xiaofei Xie, Lei Ma, Lingjun Zhou, Yang Liu, Chi Xu, and Jianjun Zhao. 2020. Cats Are Not Fish: Deep Learning Testing Calls for Out-Of-Distribution Awareness. In *FSE*.
[11] Natesh M Bhat. [n. d.]. pyttsx3: Text-to-Speech Library for Python. Online document https://pypi.org/project/pyttsx3/.
[12] Microsoft Bing. [n. d.]. Bing image search. Online document https://www.bing.com/images/trending?FORM=ILPTRD.
[13] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2019. AutoFocus: interpreting attention-based neural networks by code perturbation. In *ASE*.
[14] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, et al. 2013. API design for machine learning software: experiences from the scikit-learn project. *arXiv preprint arXiv:1309.0238* (2013).
[15] Dave Chaffey. [n. d.]. Search engine marketing statistics 2020. Online document https://www.smartinsights.com/search-engine-marketing/search-engine-statistics/.
[16] Lingjiao Chen, Matei Zaharia, and James Zou. 2022. FrugalMCT: Efficient Online ML API Selection for Multi-Label Classification Tasks. In *PMLR*.
[17] Lingjiao Chen, Matei Zaharia, and James Y Zou. 2020. Frugalml: How to use ml prediction apis more accurately and cheaply. In *Advances in neural information processing systems*, Vol. 33. 10685–10696.
[18] Xiangning Chen, Chen Liang, Da Huang, Esteban Real, Kaiyuan Wang, Yao Liu, Hieu Pham, Xuanyi Dong, Thang Luong, Cho-Jui Hsieh, et al. 2023. Symbolic discovery of optimization algorithms. *arXiv preprint arXiv:2302.06675* (2023).
[19] Dawei Cheng, Chun Cao, Chang Xu, and Xiaoxing Ma. 2018. Manifesting bugs in machine learning code: An explorative study with mutation testing. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 313–324.
[20] Jeffrey A. Clark. [n. d.]. Pillow: Python Imaging Library. Online document https://pypi.org/project/Pillow/.
[21] darude brainstorm. 2022. A clothes checking application. Online document https://github.com/paulgan98/darude-brainstorm.
[22] Kajaree Das and Rabi Narayan Behera. 2017. A survey on machine learning: concept, algorithms and applications. *International Journal of Innovative Research in Computer and Communication Engineering* 5, 2 (2017), 1301–1309.
[23] J. Deng, W. Dong, R. Socher, L. Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *CVPR*.

[24] Saikat Dutta, August Shi, Rutvik Choudhary, Zhekun Zhang, Aryaman Jain, and Sasa Misailovic. 2020. Detecting flaky tests in probabilistic and machine learning applications. In *ISSTA*.
[25] Anurag Dwarakanath, Manish Ahuja, Samarth Sikand, Raghotham M Rao, RP Jagadeesh Chandra Bose, Neville Dubash, and Sanjay Podder. 2018. Identifying implementation bugs in machine learning based image classifiers using metamorphic testing. In *ISSTA*.
[26] Yizhak Yisrael Elboher, Elazar Cohen, and Guy Katz. 2022. Neural network verification using residual reasoning. In *Software Engineering and Formal Methods: 20th International Conference, SEFM 2022, Berlin, Germany, September 26–30, 2022, Proceedings*. Springer, 173–189.
[27] Yizhak Yisrael Elboher, Justin Gottschlich, and Guy Katz. 2020. An abstraction-based framework for neural network verification. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part I 32*. Springer, 43–65.
[28] Yang Feng, Qingkai Shi, Xinyu Gao, Jun Wan, Chunrong Fang, and Zhenyu Chen. 2020. DeepGini: prioritizing massive tests to enhance the robustness of deep neural networks. In *ISSTA*.
[29] Fuzzit.dev. [n. d.]. Pythonfuzz: coverage-guided fuzz testing for python. Online document https://gitlab.com/gitlab-org/security-products/analyzers/fuzzers/pythonfuzz.
[30] Sainyam Galhotra, Yuriy Brun, and Alexandra Meliou. 2017. Fairness testing: testing software for discrimination. In *FSE*.
[31] Xinyu Gao, Yang Feng, Yining Yin, Zixi Liu, Zhenyu Chen, and Baowen Xu. 2022. Adaptive test selection for deep neural networks. In *Proceedings of the 44th International Conference on Software Engineering*. 73–85.
[32] Simos Gerasimou, Hasan Ferit Eniser, Alper Sen, and Alper Cakan. 2020. Importance-Driven Deep Learning System Testing. In *ICSE*.
[33] Alec Go, Richa Bhayani, and Lei Huang. 2009. Twitter sentiment classification using distant supervision. *CS224N project report, Stanford* 1, 12 (2009), 2009.
[34] Patrice Godefroid, Daniel Lehmann, and Marina Polishchuk. 2020. Differential regression testing for REST APIs. In *ISSTA*.
[35] Google. 2022. Google Cloud AI. Online document https://cloud.google.com/products/ai.
[36] Eric Gossett, Cormac Toher, Corey Oses, Olexandr Isayev, Fleur Legrain, Frisco Rose, Eva Zurek, Jesús Carrete, Natalio Mingo, Alexander Tropsha, et al. 2018. AFLOW-ML: A RESTful API for machine-learning predictions of materials properties. *Computational Materials Science* (2018).
[37] Qianyu Guo, Xiaofei Xie, Yi Li, Xiaoyu Zhang, Yang Liu, Li Xiaohong, and Chao Shen. 2020. Audee: Automated Testing for Deep Learning Frameworks. In *FSE*.
[38] Dave Halter. 2022. Jedi: an awesome auto-completion, static analysis and refactoring library for Python. Online document https://jedi.readthedocs.io.
[39] Fabrice Harel-Canada, Lingxiao Wang, Muhammad Ali Gulzar, Quanquan Gu, and Miryung Kim. 2020. Is Neuron Coverage a Meaningful Measure for Testing Deep Neural Networks?. In *ESEC/FSE*.
[40] HeapSortCypher. [n. d.]. A garbage classification application. Online document https://github.com/matthew-chu/heapsortcypher.
[41] Philipp Helle, Wladimir Schamai, and Carsten Strobel. 2016. Testing of autonomous systems–Challenges and current state-of-the-art. In *INCOSE international symposium*.
[42] Charles Hill, Rachel Bellamy, Thomas Erickson, and Margaret Burnett. 2016. Trials and tribulations of developers of intelligent systems: A field study. In *VL/HCC*.
[43] Qiang Hu, Yuejun Guo, Xiaofei Xie, Maxime Cordy, Mike Papadakis, Lei Ma, and Yves Le Traon. 2023. Aries: Efficient Testing of Deep Neural Networks via Labeling-Free Accuracy Estimation. (2023).
[44] IBM. 2022. IBM Watson. Online document https://www.ibm.com/watson.
[45] M Irlbeck et al. 2015. Deconstructing dynamic symbolic execution. *Dependable Software Systems Engineering* 40 (2015), 26.
[46] Md Johirul Islam, Rangeet Pan, Giang Nguyen, and Hridesh Rajan. 2020. Repairing Deep Neural Networks: Fix Patterns and Challenges. In *ICSE*.
[47] Gunel Jahangirova, Nargiz Humbatova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of Real Faults in Deep Learning Systems. In *ICSE*.
[48] Tahir Jameel, Lin Mengxiang, and Liu Chao. 2015. Automatic test Oracle for image processing applications using support vector machines. In *2015 6th IEEE International Conference on Software Engineering and Service Science (ICSESS)*. IEEE, 1110–1113.
[49] C. Jiang, S. Huang, and Z. Hui. 2018. 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)*.
[50] Misael C Júnior, Rafael AP Oliveira, Miguel AG Valverde, Marcel P Jackowski, Fátima LS Nunes, and Márcio E Delamaro. 2017. 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. 1–6.
[51] Kaggle. [n. d.]. Twitter US Airline Sentiment. Online document https://www.kaggle.com/crowdflower/twitter-airline-sentiment.
[52] Hojat Khosrowjerdi and Karl Meinke. 2018. 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*.
[53] Miryung Kim, Thomas Zimmermann, Robert DeLine, and Andrew Begel. 2016. The emerging role of data scientists on software development teams. In *ICSE*.
[54] Miryung Kim, Thomas Zimmermann, Robert DeLine, and Andrew Begel. 2017. Data scientists in software teams: State of the art and challenges. *TSE* (2017).
[55] Alina Kuznetsova, Hassan Rom, Neil Alldrin, Jasper Uijlings, Ivan Krasin, Jordi Pont-Tuset, Shahab Kamali, Stefan Popov, Matteo Malloci, Alexander Kolesnikov, et al. 2020. The open images dataset v4. *International Journal of Computer Vision* (2020), 1–26.
[56] Seokhyun Lee, Sooyoung Cha, Dain Lee, and Hakjoo Oh. 2020. Effective white-box testing of deep neural networks with adaptive neuron-selection strategy. In *ISSTA*.
[57] James R Lewis. 2018. The system usability scale: past, present, and future. *International Journal of Human–Computer Interaction* 34, 7 (2018), 577–590.
[58] Zenan Li, Xiaoxing Ma, Chang Xu, Jingwei Xu, Chun Cao, and Jian Lü. 2020. Operational Calibration: Debugging Confidence Errors for DNNs in the Field. In *ESEC/FSE*.
[59] Mikael Lindvall, Adam Porter, Gudjon Magnusson, and Christoph Schulze. 2017. Metamorphic model-based testing of autonomous systems. In *2017 IEEE/ACM 2nd International Workshop on Metamorphic Testing (MET)*.
[60] Lingle. [n. d.]. A news summary application. Online document https://github.com/AllegraChen/uofthacks6.
[61] Tilo Linz. 2020. Testing Autonomous Systems. In *The Future of Software Quality Assurance*. Springer, Cham, 61–75.
[62] Lisa-Assistant. 2021. A voice assistant application. Online document https://github.com/AlexNguyen27/lisa-assistant-gcp.
[63] Zixi Liu, Yang Feng, Yining Yin, and Zhenyu Chen. 2022. DeepState: selecting test suites to enhance the robustness of recurrent neural networks. In *Proceedings of the 44th International Conference on Software Engineering*. 598–609.
[64] Lei Ma, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Felix Juefei-Xu, Chao Xie, Li Li, Yang Liu, Jianjun Zhao, et al. 2018. Deepmutation: Mutation testing of deep learning systems. In *ISSRE*.
[65] Pingchuan Ma, Shuai Wang, and Jin Liu. 2020. Metamorphic Testing and Certified Mitigation of Fairness Violations in NLP Models.. In *IJCAI*. 458–465.
[66] Shiqing Ma, Yousra Aafer, Zhaogui Xu, Wen-Chuan Lee, Juan Zhai, Yingqi Liu, and Xiangyu Zhang. 2017. LAMP: data provenance for graph based machine learning algorithms through derivative computation. In *FSE*.
[67] Shiqing Ma, Yingqi Liu, Wen-Chuan Lee, Xiangyu Zhang, and Ananth Grama. 2018. MODE: automated neural network model debugging via state differential analysis and input selection. In *ESEC/FSE*.
[68] David Marby and Nijiko Yonskai. 2021. Pyan3: Offline call graph generator for Python 3. Online document https://github.com/davidfraser/pyan.
[69] David Marby and Nijiko Yonskai. 2022. Lorem Picsum. Online document https://picsum.photos.
[70] Phil McMinn, Muzammil Shahbaz, and Mark Stevenson. 2012. 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*. IEEE, 141–150.
[71] Microsoft. 2022. Microsoft Azure Cognitive Services. Online document https://azure.microsoft.com/en-us/services/cognitive-services.
[72] Microsoft. 2022. Visual Studio Code. Online document https://code.visualstudio.com/.
[73] Mahdi Nejadgholi and Jinqiu Yang. 2019. A study of oracle approximations in testing deep learning libraries. In *ASE*.
[74] Augustus Odena, Catherine Olsson, David Andersen, and Ian Goodfellow. 2019. Tensorfuzz: Debugging neural networks with coverage-guided fuzzing. In *ICML*.
[75] Omakase. 2022. A food manager application. Online document https://github.com/Seangottarun/Omakase.
[76] Tejit Pabari. [n. d.]. A prescription identification application. Online document https://github.com/gitika-bose/ResearchSpring2019.
[77] Brandon Paulsen, Jingbo Wang, and Chao Wang. 2020. ReluDiff: Differential Verification of Deep Neural Networks. In *ICSE*.
[78] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Deepxplore: Automated whitebox testing of deep learning systems. In *ASPLOS*.
[79] Fabio Petrillo, Philippe Merle, Naouel Moha, and Yann-Gaël Guéhéneuc. 2016. Are REST APIs for cloud computing well-designed? An exploratory study. In *ICSOC*. Springer, 157–170.

[80] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. 2019. CRADLE: cross-backend validation to detect and localize bugs in deep learning libraries. In *ICSE*.
[81] Phoenix. [n. d.]. A fire-detection application. Online document https://github.com/yunusemreemik/Phoenix.
[82] Prolific. [n. d.]. An online platform for recruiting participants. Online document https://www.prolific.co.
[83] Python. [n. d.]. ast — Abstract Syntax Trees. Online document https://docs.python.org/3/library/ast.html.
[84] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.
[85] RainorShine. [n. d.]. A weather application. Online document https://github.com/DanialKhan6312/RainorShine.
[86] Vincenzo Riccio and Paolo Tonella. 2020. Model-based Exploration of the Frontier of Behaviours for Deep Learning System Testing. In *ESEC/FSE*.
[87] RoomR. 2020. A property management application. Online document https://github.com/rodrigoHM/RoomR-Server.
[88] Mark Santolucito, Jialu Zhang, Ennan Zhai, Jürgen Cito, and Ruzica Piskac. 2022. Learning CI Configuration Correctness for Early Build Feedback. In *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, HI, USA, March 15-18, 2022*.
[89] scikit learn. [n. d.]. scikit-learn: Machine Learning in Python. Online document https://scikit-learn.org/stable/.
[90] Muzammil Shahbaz, Phil McMinn, and Mark Stevenson. 2015. Automatic generation of valid and invalid test data for string validation routines using web searches and regular expressions. *Science of Computer Programming* 97 (2015), 405–425.
[91] Arnab Sharma and Heike Wehrheim. 2020. Higher income, larger loan? monotonicity testing of machine learning models. In *ISSTA*.
[92] SIVG. [n. d.]. A video generator application. Online document https://github.com/aarenstade/speech-img-vid-generator.
[93] Soap. [n. d.]. A public opinion analysis application. Online document https://github.com/jcavejr/soap.
[94] Matthew Sotoudeh and Aditya V Thakur. 2021. Provable repair of deep neural networks. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 588–603.
[95] Evan R Sparks, Ameet Talwalkar, Virginia Smith, Jey Kottalam, Xinghao Pan, Joseph Gonzalez, Michael J Franklin, Michael I Jordan, and Tim Kraska. 2013. MLI: An API for distributed machine learning. In *ICDM*.
[96] Madhusudan Srinivasan, Morteza Pourreza Shahri, Indika Kahanda, and Upulee Kanewala. 2018. 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*. 26–33.
[97] Bing Sun, Jun Sun, Long H Pham, and Jie Shi. 2022. Causality-based neural network repair. In *Proceedings of the 44th International Conference on Software Engineering*. 338–349.
[98] Zeyu Sun, Jie M Zhang, Mark Harman, Mike Papadakis, and Lu Zhang. 2020. Automatic testing and improvement of machine translation. In *ICSE*.
[99] Yida Tao, Shan Tang, Yepang Liu, Zhiwu Xu, and Shengchao Qin. 2019. How do api selections affect the runtime performance of data analytics tasks?. In *ASE*.
[100] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *ICSE*.
[101] Saeid Tizpaz-Niari, Pavol Cernỳ, and Ashutosh Trivedi. 2020. Detecting and Understanding Real-World Differential Performance Bugs in Machine Learning Libraries. In *ISSTA*.
[102] Chengcheng Wan, Shicheng Liu, Henry Hoffmann, Michael Maire, and Shan Lu. 2021. Are Machine Learning Cloud APIs Used Correctly?. In *43th International Conference on Software Engineering (ICSE'21)*.
[103] 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)*.
[104] WanderStub. [n. d.]. An exchange conversion application. Online document https://github.com/richardjpark26/WanderStub.
[105] Jiannan Wang, Thibaud Lutellier, Shangshu Qian, Hung Viet Pham, and Lin Tan. 2022. EAGLE: creating equivalent graphs to test deep learning libraries. In *Proceedings of the 44th International Conference on Software Engineering*. 798–810.
[106] Wikidata. 2022. A free and open knowledge base. Online document https://www.wikidata.org/.
[107] Wikipedia. [n. d.]. A free encyclopedia. Online document https://en.m.wikipedia.org/.
[108] Shuzhao Xie, Yuan Xue, Yifei Zhu, and Zhi Wang. 2022. Cost Effective MLaaS Federation: A Combinatorial Reinforcement Learning Approach. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*. IEEE, 1–10.
[109] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Hongxu Chen, Minhui Xue, Bo Li, Yang Liu, Jianjun Zhao, Jianxiong Yin, and Simon See. 2018. Deephunter: Hunting deep neural network defects via coverage-guided fuzzing. *arXiv preprint arXiv:1809.01266* (2018).

[110] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. 2019. Deephunter: A coverage-guided fuzz testing framework for deep neural networks. In *ISSTA*. 146–157.
[111] Kaidi Xu, Huan Zhang, Shiqi Wang, Yihan Wang, Suman Jana, Xue Lin, and Cho-Jui Hsieh. 2020. Fast and complete: Enabling complete neural network verification with rapid and massively parallel incomplete verifiers. *arXiv preprint arXiv:2011.13824* (2020).
[112] Shenao Yan, Guanhong Tao, Xuwei Liu, Juan Zhai, Shiqing Ma, Lei Xu, and Xiangyu Zhang. 2020. Correlations between Deep Neural Network Model Coverage Criteria and Model Quality. In *ESEC/FSE*.
[113] Heiga Zen, Viet Dang, Rob Clark, Yu Zhang, Ron J Weiss, Ye Jia, Zhifeng Chen, and Yonghui Wu. 2019. Libritts: A corpus derived from librispeech for text-to-speech. *arXiv preprint arXiv:1904.02882* (2019).
[114] Fuyuan Zhang, Sankalan Pal Chowdhury, and Maria Christakis. 2020. DeepSearch: A Simple and Effective Blackbox Attack for Deep Neural Networks. In *ESEC/FSE*.
[115] Hao Zhang and WK Chan. 2019. Apricot: a weight-adaptation approach to fixing deep learning models. In *ASE*.
[116] Jialu Zhang, De Li, John Charles Kolesar, Hanyuan Shi, and Ruzica Piskac. 2022. Automated Feedback Generation for Competition-Level Code. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*.
[117] Jialu Zhang, Ruzica Piskac, Ennan Zhai, and Tianyin Xu. 2021. Static detection of silent misconfigurations with deep interaction analysis. In *OOPSLA*.
[118] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. 2018. DeepRoad: GAN-based metamorphic testing and input validation framework for autonomous driving systems. In *ASE*. IEEE, 132–142.
[119] Xiyue Zhang, Xiaofei Xie, Lei Ma, Xiaoning Du, Qiang Hu, Yang Liu, Jianjun Zhao, and Meng Sun. 2020. Towards characterizing adversarial defects of deep learning software from the lens of uncertainty. In *ICSE*.
[120] Yuhao Zhang, Luyao Ren, Liqian Chen, Yingfei Xiong, Shing-Chi Cheung, and Tao Xie. 2020. Detecting Numerical Bugs in Neural Network Architectures. In *ESEC/FSE*.
[121] Xinghan Zhao and Xiangfei Gao. 2018. An ai software test method based on scene deductive approach. In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 14–20.
[122] Hong Zhu, Dongmei Liu, Ian Bayley, Rachel Harrison, and Fabio Cuzzolin. 2019. Datamorphic testing: A method for testing intelligent applications. In *2019 IEEE International Conference On Artificial Intelligence Testing (AITest)*.