When to Say What: Learning to Find Condition-Message Inconsistencies

Islem Bouzenia
University of Stuttgart
Germany
fi_bouzenia@esi.dz

Michael Pradel
University of Stuttgart
Germany
michael@binaervarianz.de

***Abstract*—Programs often emit natural language messages, e.g., in logging statements or exceptions raised on unexpected paths. To be meaningful to users and developers, the message, i.e., *what* to say, must be consistent with the condition under which it gets triggered, i.e., *when* to say it. However, checking for inconsistencies between conditions and messages is challenging because the conditions are expressed in the logic of the programming language, while messages are informally expressed in natural language. This paper presents CMI-Finder, an approach for detecting *condition-message inconsistencies*. CMI-Finder is based on a neural model that takes a condition and a message as its input and then predicts whether the two are consistent. To address the problem of obtaining realistic, diverse, and large-scale training data, we present six techniques to generate large numbers of inconsistent examples to learn from automatically. Moreover, we describe and compare three neural models, which are based on binary classification, triplet loss, and fine-tuning, respectively. Our evaluation applies the approach to 300K condition-message statements extracted from 42 million lines of Python code. The best model achieves a precision of 78% at a recall of 72% on a dataset of past bug fixes. Applying the approach to the newest versions of popular open-source projects reveals 50 previously unknown bugs, 19 of which have been confirmed by the developers so far.**

I. Introduction

Programs often emit natural language messages to inform the user about a specific event or an error occurring during the execution. These messages range from being purely informational, e.g., when logging the state of the program, to explaining why the entire execution gets terminated, e.g., when raising an exception. Code that triggers a message is typically guarded by some condition. To be meaningful to users and developers, the condition must match the message emitted by a program. In other words, *what* the program is saying should be consistent with *when* the program is saying it.

Unfortunately, not all condition-message pairs are consistent, which may harm the robustness of a program and make debugging unnecessarily difficult. There are two main reasons for condition-message inconsistencies. First, the condition may not accurately reflect when the developer intends to print a message or raise an exception. An incorrect condition may cause some noteworthy state to remain unnoticed, or perhaps even worse, cause an exception to be raised even though no unexpected state was ever reached. For example, consider the real-world example in Figure 1a. The condition is equivalent to `not (len(bits) == 4 and len(bits) == 6)`, which will always evaluate to `True`, while the message states that the reason for the exception is that `len(bits)` is not among the values `(4,6)`.

```
if len(bits) != 4 or len(bits) != 6 :
  raise template.TemplateSyntaxError("%r takes exactly
      four or six arguments (second argument must be '
      as')" % str(bits[0]))
```

(a) Inconsistency in the Pinax project due to an incorrect condition.

```
if n2 > n1 :
  raise ValueError('Total internal reflection impossible
       for n1 > n2')
```

(b) Inconsistency in the Sympy project due to an incorrect message.

Fig. 1: Real-world examples of condition-message inconsistencies.

Second, the message may be incomplete, misleading, or even outright wrong. In this case, a user or developer may not understand the reason for a message or an exception, and as a result, perhaps even modify the code or the input in a way that introduces more bugs. Figure 1b shows a real-world example of this scenario. The exception message incorrectly claims the problem to be that `n1 > n2`, while the condition actually is raised when `n1 < n2`.

To better understand the importance of conditional messages, we perform a preliminary study of seven popular open-source projects written in Python. Analyzing the if-statements in their code shows that 20% of them output a message. In other words, there is a large number of conditions and exceptions that developers intend to be consistent, motivating a technique for automatically checking this property. We further analyze commits in these project to search for fixes of condition-message inconsistencies. The search results show that the inconsistency problem affects even popular open-source projects, such as scikit-learn (3 instances over 1,000 bug-fixing commits), Scrapy (2 instances over 1,000 bug-fixing commits), and Sympy (4 instances over 1,000 bug-fixing commits). In addition, 10% to 11% of the bug-fixing commits where a change occurs in a condition-message statement are fixes of condition-message inconsistencies.

Finding condition-message inconsistencies requires a semantic understanding of both the condition and the message. The problem is compounded by the fact that conditions are expressed in the formal syntax of the programming language, while the exception message is expressed in natural language, possibly interspersed with code fragments. Thus, any approach addressing this problem needs a semantic understanding of code and natural language.

This paper presents CMI-Finder, a deep learning-based approach for detecting condition-message inconsistencies. The basic idea is to extract message-printing statements and the conditions that guard them, and to feed them into a neural model that predicts whether the two are consistent. Realizing this idea leads to two main challenges. The first challenge is the training data problem. While examples of likely consistent condition-message pairs are amply available, obtaining large amounts of inconsistent examples is non-trivial. We address this challenge through six techniques to create a large and diverse set of inconsistent examples from existing code bases. The strategies range from classical mutation operators, over pattern-based recombination of consistent pairs, to using a large language model (Codex) to create realistic yet wrong messages. The second challenge is designing an effective neural model for the prediction task. We present three orthogonal formulations of this learning task—binary classification, a distance-based model trained with triplet loss, and fine-tuning a pre-trained model—and compare their effectiveness.

The most closely related prior work cross-checks natural language comments against code [1], [2], [3], [4] and translates comments to specifications [5], [6]. While code-comment inconsistency may hamper maintenance and reuse, we target inconsistencies that directly affect the runtime behavior. Other prior work points out difficulties in correctly handling exceptions [7], [8] and proposes techniques for detecting incorrect error handling code [9], [10], [11], as well as predicting error handling code [12], [13]. While these approaches are about handling exceptional messages, we here address the problem of correctly reporting such messages. CMI-Finder also relates to existing approaches on deep learning-based bug detection [14], [15], [16], [17], [18], [19]. Our approach contributes by addressing a previously overlooked kind of problem, by proposing a distance-based model trained with triplet loss, which differs from the binary classification models used in the past, and by addressing the training data problem through six data generation strategies.

Our evaluation applies CMI-Finder to 42 million lines of Python code from popular open-source projects. We find that the approach effectively finds condition-message inconsistencies, e.g., with a precision of 78% and a recall of 72% when applied to a dataset of 66 examples extracted from past, real-world bug fixes. Checking the newest versions of widely used projects with CMI-Finder, about one out of three warnings by CMI-Finder is a true positive, which allowed us to detect 50 previously unknown real-world bugs, some of which the developers have already confirmed. A comparison with a popular linter and an existing neural model shows that CMI-Finder complements and outperforms them by finding otherwise missed problems.

In summary, this paper contributes the following:

- We are the first to address the problem of detecting condition-message inconsistencies.
- We present six techniques for generating a large and diverse set of likely inconsistent condition-message pairs to be used as training data.
- We present three neural models that formulate the inconsistency detection problem in different ways and compare their effectiveness.
- We provide empirical evidence that the approach successfully finds bugs fixed in the past and reveals previously unknown bugs in widely used projects.

II. Problem Statement

Before describing our approach in detail, the following defines the problem we are addressing and explains what is challenging about it. CMI-Finder reasons about conditional statements that throws a warning message, defined as follows:

Definition 1 (Condition-message pair)**.** A condition-message pair (c, m) consists of

- a boolean expression c that represents the condition for triggering a message, and
- a statement m that emits a message to the user when c evaluates to true.

For example, consider the following code:

```
if condition:
    // possibly some other code here
    raise ExceptionType("message")
```

The corresponding condition-message pair (c, m) is:

$$(\texttt{condition}, \texttt{ExceptionType("message")})$$

The goal of our approach is to check whether a condition-message pair is consistent. Nevertheless, such statement can be consistent, inconsistent or neutral.

Definition 2 (Inconsistency)**.** A condition-message pair (c, m) is *inconsistent* if and only if c and the condition described by m cannot be true at the same time, i.e., they contradict each other.

We call an inconsistent pair (c, m) a *condition-message inconsistency*. Note that the above definition is rather strict about what to consider an inconsistency. In particular, it does not require that c and m logically imply each other, but only that they clearly contradict each other. The reason is that some condition-message pairs in real-world code are only weakly related, e.g., due to generic messages, such as "error", or strings printed to produce well-formatted output.

The goal of CMI-Finder is to automatically detect condition-message inconsistencies. Once an inconsistency has been identified, the developers can decide which part of the pairs to change to improve the code. The key challenge for detecting

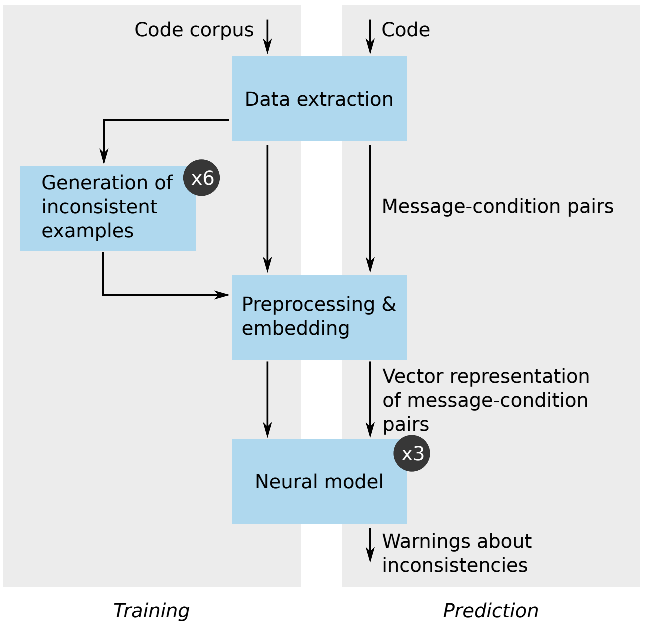

Code corpus
Code
Data extraction
Generation of inconsistent examples
x6
Message-condition pairs
Preprocessing & embedding
Vector representation of message-condition pairs
Neural model
x3
Warnings about inconsistencies
Training
Prediction

Fig. 2: Overview of the approach.

inconsistent condition-message pairs (c, m) is that, while c has precise semantics defined by the programming language, m typically contains a message formulated in natural language. Given the inherent fuzziness of natural language, accurately checking a condition and a message are logically contradictory is practically impossible. Instead, CMI-Finder is based on neural networks, which have been shown to be highly effective at reasoning about fuzzy information, e.g., natural language, associated with source code [20].

III. Approach

This section presents CMI-Finder, an automated, neural network-based approach for detecting inconsistent condition-message pairs. We start with an overview of our framework and then describe each component in detail.

A. Overview

Figure 2 gives an overview of the four main steps of our approach. As in most neural software analyses, there is a training phase, during which we train a model, and a prediction phase, during which the trained model is applied to previously unseen code. Both phases share the first step, which extracts all condition-message pairs from a given code corpus (Section III-B). During training, these condition-message pairs serve as training data, whereas during prediction, the approach tries to find inconsistencies in them. To effectively train a model that distinguishes consistent from inconsistent examples, we need large amounts of both kinds of examples. Following the common assumption that most real-world code is correct, the extracted condition-message pairs are considered to be consistent during training. To obtain inconsistent examples, CMI-Finder generates likely incorrect condition-message pairs by mutating and re-combining the pairs extracted from the code corpus (Section III-C). As the third step, the approach preprocesses all pairs, e.g., by removing duplicates during training, and then represents them as vectors suitable for neural reasoning (Section III-D). Finally, the fourth step is a neural model that learns to determine whether a condition and an exception are consistent with each other (Section III-E). Once trained, CMI-Finder applies the model to condition-message pairs extracted from previously unseen code, where it predicts the probability that a pair is inconsistent. Those pairs predicted as most likely inconsistent are then shown to a user as warnings.

CMI-Finder is designed as a framework that supports different variants of the overall approach. In particular, we present six different ways of generating inconsistent examples. Likewise, we present three different ways of formulating the machine learning task, along with suitable neural models, and compare their effectiveness.

B. Data Extraction

The first step of CMI-Finder is about extracting condition-message pairs from the given code. The approach parses the code into an abstract syntax tree and then identifies statements that emit a message and that are guarded by an `if`-condition within the same function. As message-emitting statements, we consider statements that raise an exception and function calls involving the name `print` or `log`, which we found to be common in real-world Python code during initial experiments. For each such statement, the approach extracts a pair (c, m) as described in Definition 1.

The extractor considers three types of condition-message pairs:

- *Simple conditional statements* that consist of one condition and a body, e.g.,:

```
if condition:
  // possibly some other code here
  raise ExceptionType("message")
```

- *Multi-branch statements*, which are a sequence of `if`, `elif`, and `else` branches with their corresponding bodies, e.g.,:

```
if c1:
  // some code
elif c2:
  log.warning("message")
else:
  raise Exception2("message")
```

- *Nested ifs*, which are a sequence of `if` branches that are nested inside another, e.g.,:

```
if c1:
  // some code
  if c2:
    print("message")
```

For each message-emitting statement, the extractor gathers and combines all boolean expressions guarding it. For example, for a message emitted in the `else` branch of a multi-branch statements, the associated condition is the conjunction of the negated conditions of all other branches preceding the `else` branch.

We filter, simplify, and augment the extracted pairs in three ways. First, we perform a cleaning step to exclude condition-message pairs with an empty message Second, we logically simplify conditions to make them easier to understand and to also reduce their number of tokens. For example, the condition `not a == b` will be simplified into `a != b`. Third, we inline any strings literals stored in variables into the exception message, if a string literal is assigned to the variable within the scope of the if-statement.

C. Generation of Inconsistent Examples

The second step of CMI-Finder is to generate likely inconsistent condition-message pairs based on the likely consistent pairs extracted from the training code corpus. This step is crucial for the overall effectiveness of the approach, as the data used to train a neural model determines what it will be able to find. When generating inconsistent examples, we pursue three goals: realism, diversity, and scalability. First, the inconsistent examples should be *realistic*, in the sense that the approach should create mistakes that developers might actually make. Realism is important because we want the trained model to ultimately detect real-world bugs, instead of learning to distinguish between artificial and real condition-message pairs. Second, the inconsistent examples should be *diverse*, i.e., cover a wide range of different kinds of inconsistencies. Diversity is important to enable the approach to find many different kinds of problems. Third, our techniques to generate inconsistent examples must be *applicable at scale*, i.e., yield many thousands of examples with reasonable effort. Scalability matters because training an effective neural model typically requires a large-scale dataset.

Since a single technique for generating inconsistent examples is unlikely to meet all the above goals, we design CMI-Finder as a framework that supports an extensible set of generation techniques. The following described the six currently supported techniques, roughly ordered by increasing complexity. Table I illustrates them with examples.

1) Mutation of Operators: Inspired by code transformations applied in mutation testing [21], this technique modifies logical, relational, and arithmetic operators in condition-message pairs. We mutate such operators either in the condition or in message, e.g., by replacing >= by <=, as shown in row 1 of Table I. All possible mutations are stored in pairs of the form $(op1, op2)$, which means that we can mutate $op1$ to $op2$ if $op1$ is in the condition and vice-versa. The list of all mutation pairs is as follows: $(==, !=), (<, >=), (>, <=), (all, any), (and, or), (not\ in, in), (is, is\ not), (not,\ .)$. The approach randomly picks from all operators in a pair and replaces one of them. In case there is no operator in the entire condition-message pair, the technique performs a logical negation by adding (or removing) the keyword `not` to (or from) the condition, which corresponds to the last pair in the list of mutations.

2) Mutation of Error Messages: Mistakes may not only occur in the programming language fragments of an condition-message pair, but also in its natural language fragments. The following complements the above technique by mutating strings used in the message. To mutate error messages, the approach relies on two strategies. First, we gather a set of commonly used natural language descriptions of logical, relational, and arithmetic operators, such as "greater than", and map them to alternatives that modify the semantics, such as "less than" or "equal". Second, we use NLTK and its WordNet interface to replace adjectives and verbs with their antonyms, e.g., replacing "invalid" with "valid", as in row 2 of Table I.

3) Random Re-combination: Copying and pasting code is a common cause of programming mistakes, and in particular, may lead to inconsistent condition-message pairs. To create such mistakes in the training dataset of CMI-Finder, we randomly recombine condition-message pairs. To this end, the approach picks two pairs, (c_1, m_1) and (c_2, m_2), and then recombines them into (c_1, m_2) and (c_2, m_1). Row 3 of Table I shows an example of an inconsistency generated by random re-combination.

4) Pattern-based Mutation: This technique matches two pairs that have a similar structure and then re-combines them. The rationale is that structurally similar pairs may be easily confused by developers. To identify two pairs as structurally similar, CMI-Finder parses the code into an AST and then abstracts specific tokens with an abstraction process. Specifically, the approach abstracts tokens as follows: (i) All identifiers are replaced with `ID`. (ii) All numeric, boolean, and string literals are replaced with `NUM`, `BOOL`, and `STR`, respectively. (iii) All comparison and negation operators are replaced with `OP`. (iv) Special built-in functions of Python, such as `len`, are replaced by `SPECF`. (v) Iterable literals, such as a tuple `(12, 8)` are replaced by `EITR`. All other tokens remain the way they are. After the token abstraction, we call the resulting pair a template. For example, consider the following condition-message pair:

```
if index == -1:
    raise forms.ValidationError('Could not find the item
        %s', % item)
```

The approach abstracts the example into this template:

```
if ID OP NUM:
    raise ID.ID(STR, % ID)
```

After abstracting all condition-message pairs into templates, the technique clusters all pairs that have the same condition template, e.g., `ID OP NUM`, and then replaces the entire condition of one pair with another condition from the same cluster. Likewise, the approach replaces the message expression of one pair, e.g., `raise ID.ID(STR, % ID)`, with an message expression of the same template. Row 4 of Table I gives a complete example.

Our pattern-based mutation technique relates to prior work on creating bugs based on templates of token sequences [22], [23] and on modifying code by replacing one code fragment with another that has the same AST node type [24]. To the best of our knowledge, we are the first to adapt these ideas to the problem of creating training data for learning-based inconsistency detection.

TABLE I: Examples of inconsistent condition-message pairs generated by our six techniques.

Id	Generation technique	Consistent example	Inconsistent example
1	Mutation of operators	`if idx >= size:` `raise ValueException("Index out of bounds")`	`if idx <= size:` `raise ValueException("Index out of bounds")`
2	Mutation of error messages	`if result.status in (0, 3):` `log.warning("Invalid status")`	`if result.status in (0, 3):` `log.warning("Valid status")`
3	Random re-combination	`if self.is_alive():` `print("Thread still running when test done")`	`if self.is_alive():` `print('subok=True not supported.')`
4	Pattern-based mutation	`if index == -1:` `raise forms.ValidationError('Could not find the item %s', % item)`	`if ellipsis_start == 1:` `raise forms.ValidationError('Could not find the item %s', % item)`
5	Embedding-based token replacement	`if not isinstance(config, (tuple, list)):` `raise TypeError('Unable to decode config: {}'.format(config))`	`if not isinstance(config, (tuple, list)):` `raise ValueError('Unable to decode config: {}'.format(config))`
6	Language model-based generation of error messages	`if x == 0:` `raise ValueError('x must not be zero')`	`if x != 0:` `raise ValueError('x cannot be lower than 0')`

5) Embedding-based Token Replacement: This technique uses a pre-trained token embedding model to replace target tokens with semantically similar tokens, which mimics mistakes caused by the common bug pattern [25] of accidentally using a wrong token. Given a condition-message pair, the approach starts by tokenizing both the condition and the message using the standard Python tokenizer. Next, it identifies a set of target tokens for replacement, which are all identifiers and operators in the pair, all literals in the condition, and all string literals in the message. We select these kinds of target tokens based on condition-message bugs observed in open-source projects. Given the set of candidates, the approach retrieves an alternative token from the pre-trained embedding model by querying for the ten nearest neighbors of the original token and by randomly picking one of them. Row 5 in Table I shows an example, where `TypeError` is replaced with the semantically similar token `ValueError`. Because string literals in error messages are typically composed of multiple words, often including some that refer to a variable used in the condition, the approach further tokenizes these strings and replaces words that match an identifier in the condition with an alternative suggested by the embedding model. For example, given the consistent example in row 5 of Table I, the approach may replace the string literal in the message with `'Unable to decode settings: {}'`, where “config” is replaced with “settings”.

6) Language Model-based Generation of Error Messages: Some semantic mutations of consistent condition-message pairs are hard to achieve using pre-defined transformations or even semantic token replacement. To complement the above techniques with more complex semantic transformations, we present a technique based on a large-scale, pre-trained language model, such as GPT-3 [26]. Trained on large code corpora, such models perform various tasks, including code completition. We here use the OpenAI Codex model[1], a descendant of GPT-3 that powers GitHub’s Copilot auto-

[1]https://openai.com/blog/openai-codex/

Algorithm 1 Language model-based generation of inconsistent examples.

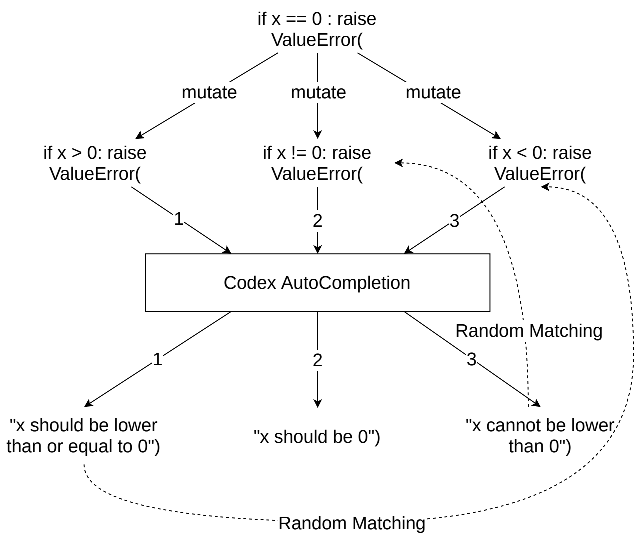
```
Input: condition-message pair $(c, m)$, pre-trained language model
    $LM$, Embedding-based token replacement generator $TRG$
Output: Set of likely inconsistent pairs $(c', m')$
    ▷ Create semantics-breaking variants of the condition:
1: $C \leftarrow$ mutate the condition $c$ using $TRG$
    ▷ Predict an error message for the modified conditions:
2: $M \leftarrow \emptyset$
3: for each $c' \in C$ do
4:     $m' \leftarrow$ ask $LM$ to complete the message expression for $c'$
5:     Add $m'$ to $M$
    ▷ Combine into likely inconsistent pairs:
6: return set of pairs $(c', m')$ randomly sampled from $C \times M$
```

completion system. Specifically, we use the ability of Codex to generate realistic error messages for a given condition.

Algorithm 1 summarizes how we use a language model to generate likely inconsistent condition-message pairs. Given a pair (c, m), the algorithm first mutates the condition c using the “token replacement” technique described in Section III-C5. Next, the algorithm queries the language model for each mutated condition to obtain a corresponding message expression. In particular, the model will generate error messages that fit the mutated condition, i.e., that likely do not fit the original condition. Given the sets of mutation conditions C and language model-generated messages M, the algorithm finally randomly combines them and returns the re-combined pairs.

An example of this generation technique is shown in Figure 3. Given a condition-message pair with condition `x == 0`, the algorithm mutates the operator, e.g., into `x > 0` and `x != 0`. It then asks Codex to complete the message for each of the mutated conditions, resulting in messages that are realistic and that likely have mutually different semantics. Randomly combining a mutated condition with a Codex-generated message hence yields a likely inconsistent condition-message pair. For our example, the algorithm picks the pair illustrated as row 6 in Table I.

Our language model-based technique relates to recent work



if x == 0 : raise ValueError(
mutate
mutate
mutate
if x > 0: raise ValueError(
if x != 0: raise ValueError(
if x < 0: raise ValueError(
1
2
3
Codex AutoCompletion
Random Matching
1
2
3
"x should be lower than or equal to 0")
"x should be 0")
"x cannot be lower than 0")
Random Matching

Fig. 3: Example of using Codex-based generation of inconsistent pairs.

on using pre-trained language models for mutation testing [27], which masks one token at a time and ask the model to predict it. In contrast to that work, our approach uses the language model to generate the entire error message, which produces more complex mutations. Moreover, we are the first to use language model-based code mutation to generate training data for training a neural bug detection model.

Each of the above techniques generates zero, one, or more likely inconsistent examples for each given consistent example. To produce a diverse training dataset, CMI-Finder first generates all possible inconsistent examples for each consistent example, and then samples a fixed number of inconsistent examples from each technique.

D. Preprocessing and Embedding

a) Cleaning and Filtering of Examples: Given the dataset of extracted and generated condition-message pairs, the next step is to prepare the dataset for neural learning. The approach performs several cleaning and filtering steps. First, we remove all duplicate examples, as those would be detrimental for training an effective model [28]. Duplicates may result from the same code appearing in multiple projects and from our generation of inconsistent examples, because some strategies may produce the same example after mutating two different examples. Second, we remove all syntactically incorrect examples, which occasionally result, e.g., from the Codex-based generated of error messages. Finally, we exclude all condition-message pairs with a condition containing more than two conjunctions, because we observe that very complex conditions tend to be hard to reason about for the model.

b) Tokenization: After cleaning the dataset, the approach represents each condition-message pair (c, m) as two sequences (C, M) of tokens. Depending on the neural model (Section III-E), we use different kinds of tokenizers. For models trained from scratch, the approach combines two tokenizers, one for the Python programming language and one for natural language. Specifically, we first tokenize both the condition and the message using Python's built-in tokenizer[2], and then further tokenize any string literal that appears in the message using the default word tokenizer of NLTK. For our third model, which fine-tunes the existing Code-T5 model, we use the default tokenizer that comes with that model, which is based on a pre-trained byte level tokenizer.

c) Embedding: Because neural models reason about numeric vectors, we must embed each condition-message pair into a fixed-size vector representation. Again, the embedding used by CMI-Finder depends on the neural model. For the models we train from scratch, CMI-Finder uses FastText because it has been shown to outperform other context-free token embedding models on code [29], while being less computationally demanding than contextual and structure-based models. We pre-train FastText on a corpus of if-conditions in Python code, which yields an embedding function that maps a token into a fixed-size vector, and then use the embedding function to map each token in (C, M) into a vector. For the Code-T5 model, embedding tokens is part of the overlap training process via an integrated embedding layer.

E. Neural Models

The fourth and final step of CMI-Finder is a neural model that predicts whether a condition-message pair (c, m) is consistent or inconsistent. We present three fundamentally different ways of formulating a learning task for this purpose. First, we use *binary classification*, i.e., a model that directly predicts whether a given condition c and message m are consistent. On a high level, this formulation is similar to existing learning-based detectors of bugs [14] and vulnerabilities [16]. Second, we use a *distance-based approach trained with triplet loss*. Intuitively, this model embeds both conditions and messages into the same vector space, so that the distance between an inconsistent condition c and message m is large. Third, we use a *text-to-text transformer* model. Unlike the two previous ones, this model is not trained from scratch but we build on a pre-trained Code-T5 model, which we fine-tune on the task of generating a label for each pair.

1) Binary Classification Model: The upper part of Figure 4 shows the components of the binary classification model. The given condition-message pair (c, m) is tokenized into two sequences (C, M) with $C = tc_1, ..., tc_k$ and $E = tm_1, ..., tm_k$. The model than concatenates these two sequences and passes them into a bi-directional, LSTM-based [30] recurrent neural module that encodes the token sequence into a vector v_{ce}. Next, a fully connected layer takes the summary vector v_{ce} as its input and predicts the probability $p_{inconsistent}$ that they are inconsistent. We train the model with binary cross-entropy, i.e., trying to nudge the model toward predicting $p_{inconsistent} = 0.0$ for the pairs extracted as-is from the code corpus and $p_{inconsistent} = 1.0$ for pairs created by our generators of likely inconsistent examples.

[2]https://docs.python.org/3/library/tokenize.html

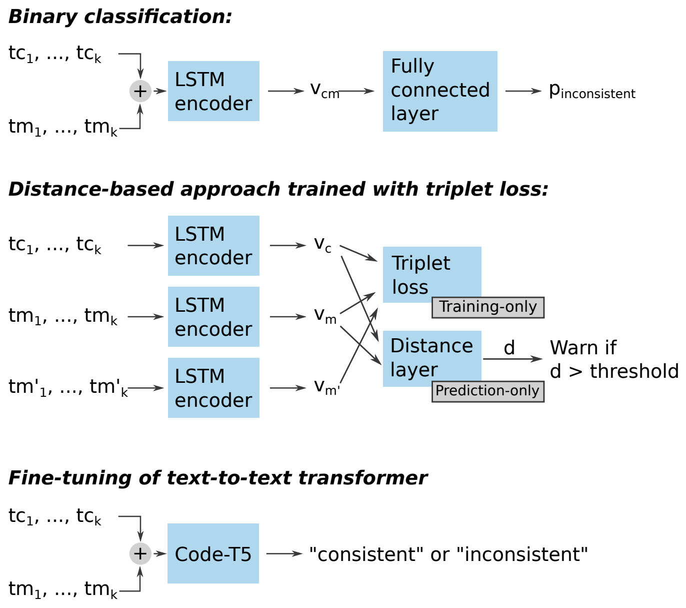


Fig. 4: Three neural models to identify inconsistencies.

2) Distance-based Approach Trained with Triplet Loss: As an alternative to binary classification, the following describes a distance-based approach that maps both conditions and messages into the same vector space. The goal of this mapping is a representation where a condition is close to its consistent message, whereas a condition and an message that are inconsistent are far away from each other. In other words, we want to minimize (maximize) the distance between the condition and the message in a consistent (inconsistent) pair.

To train a model that encodes conditions and messages in the desired way, we use contrastive learning with triplet loss [31]. The basic idea is to show three inputs at a time to the model during training: (i) an anchor input an, (ii) a "positive" input pos that should be close to the anchor, and (iii) a "negative" input neg that should not be close to the anchor. The model is then trained with triplet loss as the loss function, which implies minimizing the anchor-positive distance and maximizing the anchor-negative distance:

$$loss_{triplet} = max(dist(an, pos) - dist(an, neg) + g, 0)$$

The parameter g is the margin by which the loss tries to push the negative input away from the positive input. We compute the distance $dist$ as the squared Euclidean distance. Furthermore, we normalize the distance between two points by dividing by the average norm of the two points. For example, to normalize $dist(an, pos)$ we divide it by $(norm(an)/2 + norm(pos)/2)$.

To train the triplet-loss model, we use two kinds of triplets. On the one hand, the training data contains triplets c, m, m', where (c, m) is a consistent pair and e' is a mutation of the exception m produced by a generation technique that operates on the exception. On the other hand, we train the model with triplets c, m, c', where (c, m) is a consistent pair and c' is a mutation of the condition c produced by a technique that operates on the condition.

The middle part of Figure 4 shows the different components of the model. Before computing the triplet-loss, each component of a triplet is mapped into a vector in the same way as in the binary classifier, i.e., by applying a bi-directional LSTM-based recurrent neural network to the tokenized code. Once the model is trained with triplet-loss, the approach identifies inconsistent pairs as those that have a distance above some threshold. To this end, only two inputs – a condition c and an exception m – are passed into the model during prediction. A distance layer then compares the vector representations v_c and v_m. If the distance $dist(v_c, v_m)$ is above a configurable threshold, the approach reports the condition-message pair as inconsistent.

3) Text-to-Text Transformer: Instead of training a model from scratch, the third model builds on an already pre-trained text-to-text transformer model [32], called Code-T5 [33]. The model is pre-trained for general code understanding and generation tasks on the CodeSearchNet data set [34] and a C/C# dataset collected from BigQuery [3]. To fine-tune Code-T5 for our inconsistency detection task, we construct a dataset of pairs $(c \oplus m, l)$, where $c \oplus m$ is the concatenation of a condition and a message, and l is the corresponding label, namely "inconsistent" or "consistent". We then fine-tune the model in a supervised way to teach it to generate the expected label for a given condition-message pair.

IV. Implementation

Our implementation uses LibCST to parse the code, extract condition-message pairs, and create inconsistent pairs. We crop the tokenized condition and message to 32 tokens each, which covers more than 95% of the dataset. We pre-train FastText on 6.5 million if-statements (not necessarily an condition-message pair) with the following parameters: vector size = 32, epochs = 300, window = 10, min frequency = 3. The model is used both for replacing tokens (Section III-C5) and to embed the token sequences (Section III-D). The language model-based generation technique uses the Davinci Codex model from OpenAI through their API, with a zero temperature and a maximum number of tokens of 64. The neural models are implemented with Keras and TensorFlow as the backend. The binary classifier uses two layers of 64 biLSTM cells connected to a fully connected layer of size 128, a ReLu activation function, and a final sigmoid node to produce a probability. The triplet model uses the same kind of encoder. Both models use the Adam optimizer for training, with a learning rate of 0.0025 for the binary classifier and a decaying learning rate scheduled every 20 epochs for the triplet-loss model. Finally, we use Code-T5-small [4], which has 60M parameters. We run all our experiments on a Linux machine with 250G of memory, 48 CPU cores, and a 16G NVIDIA GPU (Tesla P100).

[3]https://console.cloud.google.com/marketplace/details/github/github-repos
[4]https://huggingface.co/Salesforce/codet5-small

V. Evaluation

To evaluate our approach, we address the following research questions:

- RQ1: How effective is CMI-Finder at detecting condition-message inconsistencies?
- RQ2: How does CMI-Finder compare to existing techniques for finding coding issues?
- RQ3: How efficient is CMI-Finder?
- RQ4: How do the hyperparameters of the models impact effectiveness?

A. Experimental Setup

a) Training data: The extractor collects 310K condition-message pairs from 40K Python projects on GitHub, of which we keep 10K for testing. To create a balanced training data set, we generate 300K inconsistent pairs, where each strategy contributes 50K pairs. Moreover, we create a dataset of 620K triplets from mixed data to train the triplet-based model.

b) Synthetic test data: We use the held-out 10K consistent pairs to create a synthetic test dataset by applying our six generation strategies.

c) Past bug fixes: As a realistic test dataset, we extract fixes of condition-message inconsistencies from version histories. To this end, we select commits that change a condition-message pair and then manually inspect the old and the new version to identify changes that fix an inconsistent pair. More specifically, we follow these steps:

1) Randomly sample popular Python projects.
2) For each project, select commits containing the keywords “bug” or “fix”.
3) Keep only commits with a change in a condition-message statement.
4) Inspect up to 1,000 randomly sampled commits per project.
5) One author selects changes that fix an inconsistency.
6) Discuss the collected changes with another author and keep only those agreed to be inconsistencies.

The resulting set of past bug fixes contains 33 pairs of consistent and inconsistent condition-message pairs belonging to 25 different projects, i.e., 66 condition-message pairs in total. We make sure to exclude these projects from the training data.

d) New projects: To evaluate the effectiveness of CMI-Finder at finding previously unknown bugs, we collect seven further Python projects from GitHub: TensorFlow, Azure, Scipy, Scrapy, Sympy, Scikit-learn, and Django. The seven repositories have 520K LoC in total, from which we extract 9,913 condition-message pairs.

B. RQ1: Effectiveness of the Approach

After training the models, we measure their effectiveness on the synthetic test data, the past bug fixes, and the previously unseen real-world projects. For the first two sets (synthetic and past bug fixes), we compute the Receiver Operating Characteristic (ROC) curve, which visualizes the tradeoff between

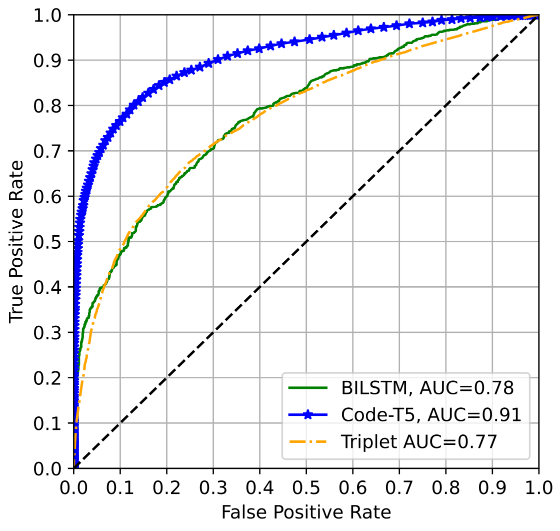


Fig. 5: ROC curves of the three models on synthetic data.

false positive rate (x-axis) and true positive rate (y-axis) for different classification thresholds. The area under the ROC curve (AUC) summarizes the ROC curve in one score, where 1 means a perfect model and 0.5 shows that the model is similar to a random predictor. The threshold values are between 0 and 1. Since the triplet model outputs the distance between two points, which is not necessarily between 0 and 1, we use quantiles to ensure that the thresholds of both models are on the same scale. Each threshold is 1/T quantile of the highest value scored by instances of the test set, where T=1000 is the number of thresholds. For the Code-T5 model, we use the probability of the output token as a score between 0 and 1.

a) Effectiveness on Synthetic Data: The ROC curve, in Figure 5, shows that the fine-tuned Code-T5 with an AUC of 0.91 outperforms the two other models, which have an AUC of 0.78 and 0.77, respectively. At a false positive rate (FPR) of 0.2, the Code-T5-based model reaches an F1-score of 0.83, with 0.86 precision and 0.81 recall.

b) Effectiveness on Past Bug Fixes: After evaluating our model on synthetic data, we perform a complementary evaluation on the dataset of 66 examples extracted from past fixes of real-world bugs. Figure 6 illustrates the ROC curve obtained by the models. Again, the Code-T5 model performs better than the BILSTM and triplet-based models (AUC of 0.82 vs. 0.55 and 0.53). Moreover, the AUC of the Code-T5 on this test set is closer to its AUC on the synthetic data, demonstrating that the model generalizes better to real data. In contrast, the AUC of the BILSTM and triplet models drops considerably, from 0.78 on synthetic data to 0.55 and 0.53 on real data. At an FPR of 0.2, the Code-T5 model achieves an F1 score of 0.75, corresponding to 0.78 precision and 0.72 recall.

c) Effectiveness on Real-World Projects: Next, we apply our approach to all condition-message pairs in the previously unseen projects. As a result, we get a prediction score for each pair, and sort the pairs by their predicted inconsistency. To get

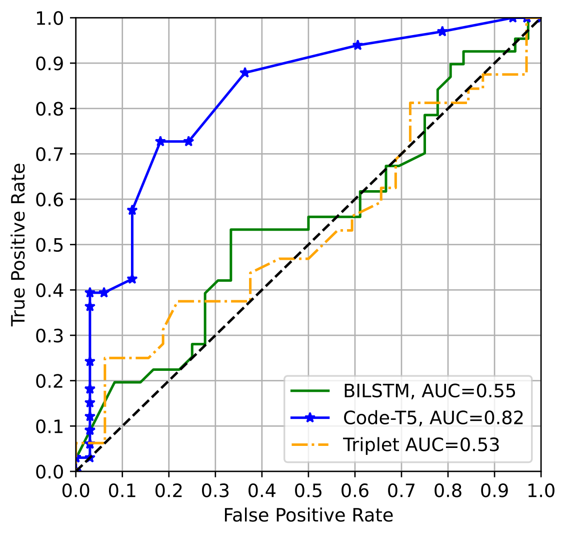


Fig. 6: ROC curves of the three models on past bug fixes.

an estimate of the true positive rate on real-world projects, we systematically inspect the top-ranked warnings and label them as true positives and false positives according to Definition 2. To reduce potential bias, the two authors discuss classification until they reach a consensus. Finally, for each true positive, we localize the problem (i.e., condition or message) and suggest a fix by submitting a pull request to the project.

Because the ratio of inconsistent to consistent pairs is likely imbalanced and much smaller then the 50:50 ratio in the dataset of past bug fixes, it is expected to see more false positives in this setup. We sample a total of 70 warnings by sampling the top-50 warnings in exception raising statements, the top-10 warnings in print statements, and the top-10 warnings in log statements, as reported by the Code-T5 model across the seven projects. The 50-10-10 sampling roughly matches the distribution of condition-message pairs across the three kinds of message-emitting statements the approach extracts (Section III-B). Our manual checking includes, if necessary, consulting the source code to understand the context and then deciding the pair's consistency. On average, we take less than 20 seconds to check pairs that do not require context and up to one minute to check context-requiring pairs.

Among the 70 inspected pairs, we find a total of 21 true positives, i.e., about one out of three warnings reported by CMI-Finder is relevant to developers. Among the 21 true positives, 16 inconsistencies are in raise statements, two in print statements, and three in log statements.

Throughout different experiments with variants of the models, including the inspection of the 70 pairs described above, we found a total of 50 previously unknown condition-message inconsistencies. In 45 statements, the inconsistency is due to a wrong message, and in the other five it is due to a wrong condition. So far, we reported 40 of them to the developers, of which 19 have already been confirmed as bugs. Table II shows representative examples of the warnings reported by CMI-Finder. The first example combines two common causes for inconsistencies, namely a message that clearly does not match the condition and an incorrect exception type. The second example is representative for another common pattern, namely a message that describes a part of the condition, but fails to cover all reasons why a message gets triggered. The third example is another case of a message that seems unrelated to the condition. Finally, the last example is a false positive causes by the fact that the approach does not reason about data-flow relationships between variables. Providing more non-local information to the model could help avoid such false positives in the future.

Finding 1: The approach reveals a total of 50 previously unknown bugs in widely used projects, has an F1-score of 0.75 on a balanced dataset of past bug fixes, and a true positive rate of one out of three when being applied to real-world projects.

C. *RQ2: Comparison with Baselines*

We compare CMI-Finder to two existing techniques for detecting coding issues. First, we compare with flake8, which is a pattern-based linter popular among Python developers[5]. Applying flake8 to the dataset of past bug fixes used in RQ1 reveals none of the inconsistent pairs, confirming our hypothesis that CMI-Finder addresses a problem that is difficult to address with a pattern-based approach. Second, we compare with a GPT3 [26]-based neural model for detecting bugs in Python[6] The model is fine-tuned by OpenAI on detecting and fixing bugs in Python code. Applied to the dataset of past bug fixes, GPT3 achieves an F1-score of 62%, which is clearly worse than the 75% achieved by CMI-Finder. Moreover, GPT3 fails to reveal any of previously unknown inconsistencies detected by CMI-Finder. We conclude from these results that training a neural model for a specific kind of bug is beneficial.

Finding 2: Compared to a widely used linter and a neural model for detecting bugs in Python, CMI-Finder is clearly more effective at finding condition-message inconsistencies.

D. *RQ3: Efficiency in Training and Prediction*

We evaluate the time taken for training the models in CMI-Finder and for predicting inconsistencies once a model has been trained. Since the models differ in size and architecture, their efficiency also differs. For training, the BILSTM model (165K parameters) takes 58 seconds per epoch with a batch size of 512, the triplet-based model (192K parameters) takes 42 seconds per epoch with a batch size of 1,024, and the Code-T5 model (60M parameters) takes 3 hours and 25 minutes per epoch with a batch size of 32. The size of the training data is the same in the three setups. The batch size differs based on the data our GPU can load and operate on

[5]https://flake8.pycqa.org
[6]https://beta.openai.com/playground/p/default-fix-python-bugs

TABLE II: Real-world problems identified by CMI-Finder (RQ3).

Id	Instance	Project	Status	Comment
1	`if not isinstance(p2, PolyElement): raise ValueError('p1 and p2 must have the same ring')`	Simpy	Confirmed	The message is claiming the problem to be p1 and p2 not having the same ring, however the condition is testing if p2 is a polynomial. The accepted fix was "p2 must be a polynomial". Also the exception type is wrong.
2	`if not (os.path.isdir(tf_source_path) and os.path.isfile(syslibs_configure_path) and os.path.isfile(workspace0_path)): raise ValueError('The path to the TensorFlow source must be passed as the first argument')`	TensorFlow	Confirmed	The message reflects the first part of the condition only, and hence, does not imply the entire condition.
3	`if len(index_keys) > 1: tf_logging.warning("SparseFeature is a complicated feature config and should only be used after careful consideration of VarLenFeature.")`	TensorFlow	To be reported	Message does not reflect any part of the condition.
4	`if self._name is None and self._values is not None: raise ValueError("At least one of name (%s) and default_name (%s) must be provided."% (self._name, self._default_name))`	TensorFlow	False positive	The model reports a warning because the condition checks `name` but the message refers to `default_name`. However, the statement is correct because in another method `default_name` gets assigned to `name` if `name` is `None`.

simultaneously. Once the one-time effort of training a model is done, prediction is much faster. The BILSTM model performs 3,000 predictions per second when run on the GPU and 600 predictions per second on the CPU. The triplet model performs 1,500 predictions per second on the GPU and 500 predictions per second on the CPU. Finally, Code-T5 performs 40 predictions per second on the GPU and 16 predictions per second on the CPU. Compared to the other two models, Code-T5 is less efficient, but as it clearly provides the best results and has prediction times that are fast enough for practical use, we consider it to be the default model of CMI-Finder.

Finding 3: Depending on the model, training takes between several minutes and several hours. Once a model is trained, tens to thousands of predictions can be performed within a second.

E. RQ4: Impact of Hyperparameters

To understand how our results depend on hyperparameters of the neural models, we experiment with several configurations. For CodeT5, we use the defaults of the model, except for the number of epochs and the batch size during fine-tuning. Our default CodeT5 model is trained for two epochs. However, experiments show that training for another epoch helps increase the AUC score on past bug fixes from **0.82** to **0.84**, whereas training for only one epoch lowers the AUC score to **0.78**, as shown in Figure 7.

For the BiLSTM model, we vary the number of layers, the size of each layer, and the dropout value. Our default model has two BiLSTM layers of size 64, each followed by a feed-forward layer of 128 nodes. To investigate the sensitivity of the model to these parameters, we vary each of the three parameters while keeping the others at their default value. Specifically, we vary the number of BiLSTM layers (variants called 64_64_64 and 64), the size of the BiLSTM layers (variants called 128_128 and 32_32), and the size of the feed-forward layer (variants called dense_64 and dense_256). Figure 8 summarizes the results of these experiments. The figure shows that changing the given parameters has a relatively little effect on the AUC score (less than 0.08). Our default model is the best in terms of AUC on past bug fixes.

We also investigate the triplet model's sensitivity to hyper-parameters. Specifically, we vary the distance metric between squared Euclidean (default), Euclidean distance, and Manhattan distance. We also vary the parameters of the loss formula (default: $\lambda_1 = \lambda_2 = g = 1$):

$$loss_{trip} = max(\lambda_1 * dist(an, pos) - \lambda_2 * dist(an, neg) + g, 0)$$

Figure 9 summarizes the results of our experiments. The figure shows that changing the given parameters has a relatively small effect on the AUC score (less than 0.03 of difference). Our default model is the best in terms of AUC on past bug fixes.

Finding 4: The effectiveness of the models is relatively stable w.r.t. changes of their hyperparameters.

VI. Threats to Validity

We implement CMI-Finder for Python, and even though the problem of condition-message inconsistencies is not Python-specific, our results may not generalize to other languages. Our approach for generating and labeling training data assumes that the condition-message pairs extracted from open-source projects are consistent, which may not always hold. As a result, we cannot guarantee that the generated inconsistent pairs are indeed all inconsistent. To avoid biasing our evaluation toward synthetically generated examples, we also evaluate on past bug fixes and previously unseen projects, which has been shown to

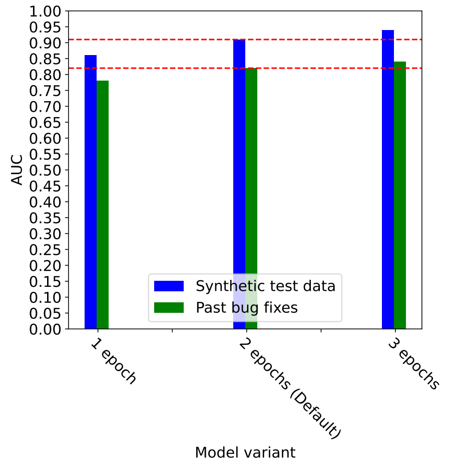


Fig. 7: AUC value for CodeT5 variants on both synthetic data and past-bug fixes.

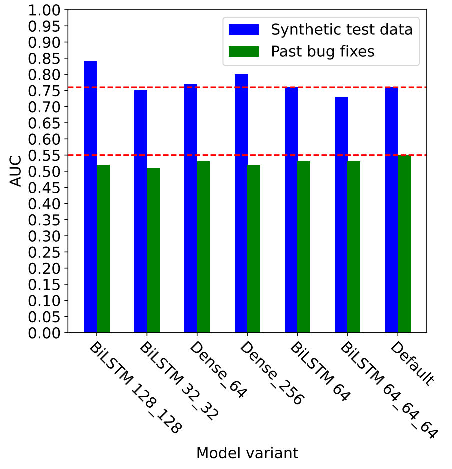


Fig. 8: AUC value for BiLSTM variants on both synthetic data and past bug fixes.

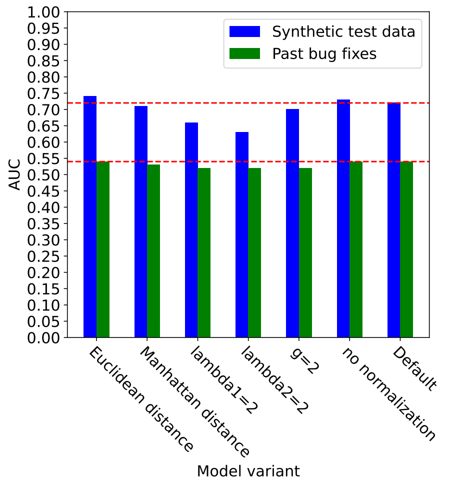


Fig. 9: AUC value for Triplet model variants on synthetic data and past bug fixes.

be important [17]. Our implementation focuses on condition-message statements with up to two conjunctions (e.g., `if x and y and z`), which covers 91% of all conditions in our dataset. Finally, we collect the dataset of past bug fixes by hand, which poses a risk of potential bias, which we try to mitigate by discussing all examples among the authors until reaching an agreement.

VII. Related Work

Natural language vs. code: Our work is part of a larger stream of work on cross-checking natural language against code. Prior work compares natural language comments against method-level code fragments [1], [2], [3], [4] and translates comments to formal specifications [35], [5], [6]. A unique property of condition-message inconsistencies is to directly affect the execution behavior of a program.

Exception handling: Exceptions thrown by third-party APIs and exception handling code have been found to be a common cause of bugs [7], [8]. Several approaches for detecting exception-related bugs have been proposed [36], [9], [10], [11]. Another line of work predicts what code to surround with a try-block and what the exception handling code should be [12], [13]. CMI-Finder complements prior work, as we are the first to address condition-message inconsistencies.

Missing conditions: One cause for condition-message inconsistencies are missing conditions. Chang et al. [37] address this problem by mining program dependency graphs to find conditions that should be checked. In a similar vein, frequent itemset mining can also be used to find missing conditions [38]. These approaches use data mining and focus on conditions to check before API calls, instead of using deep learning to check conditions that guard messages.

Learning-based bug detection: CMI-Finder is an instance of learning-based bug detection, which has received significant attention in recent years. Existing approaches check for name-related bugs [14], variable misuse bugs [15], security vulnerabilities [16], [17], [18], and misleading variable names [19]. All of these approaches are based on binary classification. CMI-Finder contributes to the state of the art by addressing a kind of bug not considered by previous work, by providing six techniques for generating buggy examples, and by comparing three different ways of formulating the learning task.

Neural models of code: An recurring question in neural software analysis [20] is how to embed code into a vector representation, and approaches based on AST paths [39], control flow graphs [40], ASTs [41], and a combination of token sequences and graphs [42] have been proposed. Our models build on a token sequence-based encoding because both the conditions and the exceptions are relatively short fragments of code. Problems addressed by neural models of code include making predictions about code changes [43], [44], program repair [45], [46], code completion [47], [48], [49], code search [50], [51], and type prediction [52], [53], [54], [55]. Similar to our distance-based model, a type prediction model by Allamanis et al. [56] also uses triplet loss, in their case to bring symbols that have the same type close to each other in an embedding space.

VIII. Conclusion

This paper presents a framework for learning to find condition-message inconsistencies. Powered by six data generation techniques and three neural models, the approach learns to warn about inconsistent statements. Evaluating the approach on Python shows that it is effective at finding bugs developers have fixed in the past, and even finds 50 previously unknown problems in real-world projects.

Data Availability

The code and data for this work are publicly available:

https://doi.org/10.5281/zenodo.7577795

Acknowledgments

This work was supported by the European Research Council (ERC, grant agreement 851895), and by the German Research Foundation within the ConcSys and DeMoCo projects.

References

[1] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/*icomment: bugs or bad comments?*/," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, T. C. Bressoud and M. F. Kaashoek, Eds. ACM, 2007, pp. 145–158. [Online]. Available: https://doi.org/10.1145/1294261.1294276

[2] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@tcomment: Testing javadoc comments to detect comment-code inconsistencies," in *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*, G. Antoniol, A. Bertolino, and Y. Labiche, Eds. IEEE Computer Society, 2012, pp. 260–269. [Online]. Available: https://doi.org/10.1109/ICST.2012.106

[3] I. K. Ratol and M. P. Robillard, "Detecting fragile comments," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, 2017, pp. 112–122.

[4] S. Panthaplackel, J. J. Li, M. Gligoric, and R. J. Mooney, "Deep just-in-time inconsistency detection between comments and source code," in *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*. AAAI Press, 2021, pp. 427–435. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/16119

[5] A. Blasi, A. Goffi, K. Kuznetsov, A. Gorla, M. D. Ernst, M. Pezzè, and S. D. Castellanos, "Translating code comments to procedure specifications," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, F. Tip and E. Bodden, Eds. ACM, 2018, pp. 242–253. [Online]. Available: https://doi.org/10.1145/3213846.3213872

[6] J. Zhai, Y. Shi, M. Pan, G. Zhou, Y. Liu, C. Fang, S. Ma, L. Tan, and X. Zhang, "C2s: Translating natural language comments to formal program specifications," in *FSE*, 2020.

[7] F. Ebert, F. Castor, and A. Serebrenik, "An exploratory study on exception handling bugs in java programs," *J. Syst. Softw.*, vol. 106, pp. 82–101, 2015. [Online]. Available: https://doi.org/10.1016/j.jss.2015.04.066

[8] R. Coelho, L. Almeida, G. Gousios, and A. van Deursen, "Unveiling exception handling bug hazards in android based on github and google code issues," in *12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015, Florence, Italy, May 16-17, 2015*, M. D. Penta, M. Pinzger, and R. Robbes, Eds. IEEE Computer Society, 2015, pp. 134–145. [Online]. Available: https://doi.org/10.1109/MSR.2015.20

[9] S. Jana, Y. J. Kang, S. Roth, and B. Ray, "Automatically detecting error handling bugs using error specifications," in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, T. Holz and S. Savage, Eds. USENIX Association, 2016, pp. 345–362. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/jana

[10] Y. Tian and B. Ray, "Automatically diagnosing and repairing error handling bugs in C," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, E. Bodden, W. Schäfer, A. van Deursen, and A. Zisman, Eds. ACM, 2017, pp. 752–762. [Online]. Available: https://doi.org/10.1145/3106237.3106300

[11] Z. Jia, S. Li, T. Yu, X. Liao, J. Wang, X. Liu, and Y. Liu, "Detecting error-handling bugs without error specification input," in *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 2019, pp. 213–225. [Online]. Available: https://doi.org/10.1109/ASE.2019.00029

[12] T. Nguyen, P. Vu, and T. Nguyen, "Code recommendation for exception handling," in *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, P. Devanbu, M. B. Cohen, and T. Zimmermann, Eds. ACM, 2020, pp. 1027–1038. [Online]. Available: https://doi.org/10.1145/3368089.3409690

[13] J. Zhang, X. Wang, H. Zhang, H. Sun, Y. Pu, and X. Liu, "Learning to handle exceptions," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020.

[14] M. Pradel and K. Sen, "DeepBugs: A learning approach to name-based bug detection," *PACMPL*, vol. 2, no. OOPSLA, pp. 147:1–147:25, 2018. [Online]. Available: https://doi.org/10.1145/3276517

[15] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, 2018. [Online]. Available: https://openreview.net/forum?id=BJOFETxR-

[16] Z. Li, S. X. Deqing Zou and, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: A deep learning-based system for vulnerability detection," in *NDSS*, 2018.

[17] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?" *CoRR*, vol. abs/2009.07235, 2020. [Online]. Available: https://arxiv.org/abs/2009.07235

[18] Y. Li, S. Wang, and T. N. Nguyen, "Vulnerability detection with fine-grained interpretations," in *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, Eds. ACM, 2021, pp. 292–303. [Online]. Available: https://doi.org/10.1145/3468264.3468597

[19] J. Patra and M. Pradel, "Nalin: Learning from runtime behavior to find name-value inconsistencies in jupyter notebooks," in *ICSE*, 2022.

[20] M. Pradel and S. Chandra, "Neural software analysis," *Commun. ACM*, vol. 65, no. 1, pp. 86–96, 2022. [Online]. Available: https://doi.org/10.1145/3460348

[21] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Mutation testing advances: an analysis and survey," in *Advances in Computers*. Elsevier, 2019, vol. 112, pp. 275–378.

[22] D. B. Brown, M. Vaughn, B. Liblit, and T. W. Reps, "The care and feeding of wild-caught mutants," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, 2017, pp. 511–522.

[23] J. Patra and M. Pradel, "Semantic bug seeding: a learning-based approach for creating realistic bugs," in *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, Eds. ACM, 2021, pp. 906–918. [Online]. Available: https://doi.org/10.1145/3468264.3468623

[24] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments." in *USENIX Security Symposium*, 2012, pp. 445–458.

[25] R. Karampatsis and C. A. Sutton, "How often do single-statement bugs occur? the manysstubs4j dataset," *CoRR*, vol. abs/1905.13334, 2019. [Online]. Available: http://arxiv.org/abs/1905.13334

[26] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., 2020. [Online]. Available: https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

[27] R. Degiovanni and M. Papadakis, "μbert: Mutation testing using pre-trained language models," *CoRR*, vol. abs/2203.03289, 2022. [Online]. Available: https://arxiv.org/abs/2203.03289

[28] M. Allamanis, "The adverse effects of code duplication in machine learning models of code," *arXiv preprint arXiv:1812.06469*, 2018.

[29] Y. Wainakh, M. Rauf, and M. Pradel, "Idbench: Evaluating semantic representations of identifier names in source code," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 562–573. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00059

[30] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[31] K. Q. Weinberger and L. K. Saul, "Distance metric learning for large margin nearest neighbor classification." *Journal of machine learning research*, vol. 10, no. 2, 2009.

[32] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, pp. 140:1–140:67, 2020. [Online]. Available: http://jmlr.org/papers/v21/20-074.html

[33] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, M. Moens, X. Huang, L. Specia, and S. W. Yih, Eds. Association for Computational Linguistics, 2021, pp. 8696–8708. [Online]. Available: https://doi.org/10.18653/v1/2021.emnlp-main.685

[34] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[35] A. Goffi, A. Gorla, M. D. Ernst, and M. Pezzè, "Automatic generation of oracles for exceptional behaviors," in *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, A. Zeller and A. Roychoudhury, Eds. ACM, 2016, pp. 213–224. [Online]. Available: https://doi.org/10.1145/2931037.2931061

[36] S. Thummalapenta and T. Xie, "Mining exception-handling rules as sequence association rules," in *International Conference on Software Engineering (ICSE)*. IEEE, 2009, pp. 496–506.

[37] R.-Y. Chang, A. Podgurski, and J. Yang, "Finding what's not there: a new approach to revealing neglected conditions in software," in *International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2007, pp. 163–173.

[38] S. Thummalapenta and T. Xie, "Alattin: Mining alternative patterns for detecting neglected conditions," in *International Conference on Automated Software Engineering (ASE)*. IEEE, 2009, pp. 283–294.

[39] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 40:1–40:29, 2019. [Online]. Available: https://doi.org/10.1145/3290353

[40] Y. Wang, F. Gao, L. Wang, and K. Wang, "Learning semantic program embeddings with graph interval neural network," in *OOPSLA*, 2020.

[41] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *ICSE*, 2019.

[42] V. J. Hellendoorn, C. Sutton, R. Singh, P. Maniatis, and D. Bieber, "Global relational models of source code," in *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. [Online]. Available: https://openreview.net/forum?id=B1lnbRNtwr

[43] T. Hoang, H. J. Kang, D. Lo, and J. Lawall, "Cc2vec: Distributed representations of code changes," in *ICSE*, 2020.

[44] S. Brody, U. Alon, and E. Yahav, "A structural model for contextual code changes," in *OOPSLA*, 2020.

[45] R. Gupta, S. Pal, A. Kanade, and S. K. Shevade, "Deepfix: Fixing common C language errors by deep learning," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, S. P. Singh and S. Markovitch, Eds. AAAI Press, 2017, pp. 1345–1351. [Online]. Available: http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14603

[46] E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang, "Hoppity: Learning graph transformations to detect and fix bugs in programs," in *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. [Online]. Available: https://openreview.net/forum?id=SJeqs6EFvB

[47] G. A. Aye and G. E. Kaiser, "Sequence model design for code completion in the modern IDE," *CoRR*, vol. abs/2004.05249, 2020. [Online]. Available: https://arxiv.org/abs/2004.05249

[48] S. Kim, J. Zhao, Y. Tian, and S. Chandra, "Code prediction by feeding trees to transformers," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 150–162. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00026

[49] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, 2019. [Online]. Available: https://openreview.net/forum?id=H1gKYo09tX

[50] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, pp. 933–944. [Online]. Available: https://doi.org/10.1145/3180155.3180167

[51] S. Sachdev, H. Li, S. Luan, S. Kim, K. Sen, and S. Chandra, "Retrieval on source code: a neural code search," in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. ACM, 2018, pp. 31–41.

[52] V. J. Hellendoorn, C. Bird, E. T. Barr, and M. Allamanis, "Deep learning type inference," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, G. T. Leavens, A. Garcia, and C. S. Pasareanu, Eds. ACM, 2018, pp. 152–162. [Online]. Available: https://doi.org/10.1145/3236024.3236051

[53] R. S. Malik, J. Patra, and M. Pradel, "NL2Type: Inferring JavaScript function types from natural language information," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, 2019, pp. 304–315. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00045

[54] M. Pradel, G. Gousios, J. Liu, and S. Chandra, "Typewriter: Neural type prediction with search-based validation," in *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, 2020, pp. 209–220. [Online]. Available: https://doi.org/10.1145/3368089.3409715

[55] J. Wei, M. Goyal, G. Durrett, and I. Dillig, "Lambdanet: Probabilistic type inference using graph neural networks," in *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. [Online]. Available: https://openreview.net/forum?id=Hkx6hANtwH

[56] M. Allamanis, E. T. Barr, S. Ducousso, and Z. Gao, "Typilus: Neural type hints," in *PLDI*, 2020.