UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

BEATRIZ BEZERRA DE SOUZA

Learning to Detect Text-Code Inconsistencies with Weak and Manual Supervision

Recife

2023

BEATRIZ BEZERRA DE SOUZA

Learning to Detect Text-Code Inconsistencies with Weak and Manual Supervision

A M.Sc. Dissertation presented to the Center of Informatics of Federal University of Pernambuco in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Concentration Area: Software Engineering and Programming Languages

Advisor: Prof. Dr. Marcelo Bezerra d'Amorim

Recife

2023

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

S729l Souza, Beatriz Bezerra de
Learning to detect text-code inconsistencies with weak and manual supervision / Beatriz Bezerra de Souza. – 2023.
45 f.: il., fig., tab.

Orientador: Marcelo Bezerra d'Amorim.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2023.
Inclui referências.

1. Engenharia de software. 2. Detecção de inconsistência. I. d'Amorim, Marcelo Bezerra (orientador). II. Título.

005.1 CDD (23. ed.) UFPE - CCEN 2023-30

Beatriz Bezerra de Souza

"Learning to Detect Text-Code Inconsistencies with Weak and Manual Supervision"

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação. Área de Concentração: Engenharia de Software e Linguagens de Programação

Aprovado em: 15/02/2023

ACKNOWLEDGEMENTS

I have been having the privilege of working with research and being advised by great people since high school. Therefore, chronologically, I would like to thank Gustavo Soares and Wilza Moreira for their guidance during my first research project when I was still a high school student. Many thanks also to my excellent research advisors during my bachelors: Prof. Dr. Patrícia Machado and Prof. Dr. Rohit Gheyi.

My masters has been quite an experience. Thanks to my advisor, Prof. Dr. Marcelo D'Amorim, and my future PhD advisor, Prof. Dr. Michael Pradel, who believed, encouraged and supported my work since the beginning. I truly enjoyed working on our many research projects and deadlines. Both of you guided me and comforted me when things did not go as expected. Therefore, OBRIGADA and DANKE!

I am also very grateful for the opportunity of being a Research Intern at the Software Lab in Stuttgart during my masters. The weather may get cold in Germany, but this group is as warm as the temperatures in Northeast Brazil. I am looking forward for my PhD journey with them!

Thank you also to my flatmates, Sarah and Philipp, for answering many questions about how household chores are done in Germany, showing me some German culture - including introducing me to a Biergarten, and also for the surprises, such as when St. Nikolaus left chocolates at my room's door.

Finally, many thanks to my family for all their support. Due to COVID, we stayed most of my masters together and I asked many times for them to try to be quiet when I was going to have virtual meetings about my research. Thanks for your patience! However, I could hear the chickens even when I was at the other side of the Atlantic Ocean :)

ABSTRACT

Source code often is associated with a natural language summary, enabling developers to understand the behavior and intent of the code. For example, method-level comments summarize the behavior of a method and test descriptions summarize the intent of a test case. Unfortunately, the text and its corresponding code sometimes are inconsistent, which may hinder code understanding, code reuse, and code maintenance. We propose TCID, an approach for Text-Code Inconsistency Detection, which trains a neural model to distinguish consistent from inconsistent text-code pairs. Our key contribution is to combine two ways of training such a model. First, TCID performs weakly supervised pre-training based on large amounts of consistent examples extracted from code as-is and inconsistent examples created by randomly recombining text-code pairs. Then, TCID fine-tunes the model based on a small and curated set of manually labeled examples. This combination is motivated by the observation that weak supervision alone leads to models that generalize poorly to real-world inconsistencies. Our evaluation applies the two-step training procedure to four state-of-the-art models and evaluates it on two text-vs-code problems: 40.7K method-level comments checked against the corresponding Java method body, and—as a problem not considered in prior work—338.8K test case descriptions checked against corresponding JavaScript implementations. Our results show that a small amount of manual labeling enables the approach to significantly improve effectiveness, outperforming the current state of the art and improving the F1 score by 5% in Java and by 17% in JavaScript. We validate the usefulness of TCID's predictions by submitting pull requests, of which 10 have been accepted so far.

Keywords: inconsistency detection; pre-training; fine-tuning.

RESUMO

O código-fonte geralmente está associado a um resumo em linguagem natural, permitindo que os desenvolvedores entendam o comportamento e a intenção do código. Por exemplo, comentários em nível de método resumem o comportamento de um método e descrições de teste resumem a intenção de um caso de teste. Infelizmente, o texto e seu código correspondente às vezes são inconsistentes, o que pode atrapalhar a compreensão do código, a reutilização do código e a manutenção do código. Propomos TCID, uma abordagem para Detecção de Inconsistência de Código e Texto, que treina um modelo neural para distinguir pares de texto-código consistentes de inconsistentes. Nossa principal contribuição é combinar duas formas de treinar tal modelo. Primeiro, o TCID executa pré-treinamento fracamente supervisionado com base em grandes quantidades de exemplos consistentes extraídos do código como está e exemplos inconsistentes criados pela recombinação aleatória de pares texto-código. Em seguida, o TCID faz o ajuste fino no modelo baseado em um conjunto pequeno e curado de exemplos rotulados manualmente. Esta combinação é motivada pela observação de que a supervisão fraca por si só leva a modelos que generalizam mal a inconsistências do mundo real. Nossa avaliação aplica o procedimento de treinamento em duas etapas a quatro modelos de última geração e avalia-os em dois problemas de texto versus código: 40.7K comentários em nível de método verificados em relação ao corpo do método Java correspondente e—como um problema não considerado em trabalhos anteriores—338.8K as descrições dos casos de teste são verificadas em relação às implementações JavaScript correspondentes. Nossos resultados mostram que uma pequena quantidade de rotulagem manual permite que a eficácia da abordagem melhore significativamente, superando o estado da arte atual e melhorando a pontuação de F1 em 5% em Java e em 17% em JavaScript. Validamos a utilidade das previsões do TCID por envio de *pull requests*, dos quais 10 foram aceitos até o momento.

Palavras-chaves: detecção de inconsistência; pré-treinamento; afinamento.

LIST OF FIGURES

LIST OF LISTINGS

LIST OF TABLES

SUMÁRIO

1 INTRODUCTION

Code and natural language text co-exist in software (HINDLE et al., 2012; ERNST, 2017). Examples of this co-existence are method-level comments that summarize the behavior of a method, e.g., in the form of Javadoc or Python's docstrings. Other examples are short descriptions of test cases, which are popular in JavaScript testing frameworks; such as Jest, Jasmine, Mocha and QUnit. These frameworks use strings to document test suites and test cases. For example, consider Listing 1, where the `describe` function describes a group of test cases, and the `test` and `it` functions describe individual test cases.

Natural language descriptions associated with source code help developers to quickly understand the behavior and intent of the code, e.g., to quickly find the root cause of a failing test case. Unfortunately, textual descriptions are not always consistent with their associated code. We illustrate this problem with two concrete examples.

Listing 1 – Example of test-vs-description Inconsistency

```
describe('<UserForm />', () => {
  it(' call  handleSubmit when the form is submitted', ()=>{
    renderedComponent();
    expect(screen.getByText(/Mock ProfileContainer/i))
      .toBeInTheDocument();
  });
  ...
})
```

Source: TRACK... (2021)

Listing 2 – Example of method-vs-comment Inconsistency

```
/** @return an Iterator */
public Group[] getGroups() {
  Item[] items = super.getItems();
  Group[] groups = new Group[items.length];
  for (int i = NUM; i < groups.length; i++) {
    groups[i] = (Group) items[i];
  }
  return groups;
}
```

Source: APACHE... (2022)

Test Code-Description Inconsistency. Listing 1 also shows a UI test of the project `track` (TRACK..., 2021). This test case uses the `renderedComponent` function to render the UI on the screen and then it checks whether the text "*Mock ProfileContainer*" is present in the HTML. The description "*call handleSubmit when the form is submitted*" does *not* reflect this behavior. A closer look at the test file reveals that the likely reason for such inconsistency was copying-and-pasting: The test case declared after this one has the same description, and that test indeed calls the `handleSubmit` function upon submission of the form. A better description for the test case from Listing 1 would be "*Check if Mock ProfileContainer is present in DOM*".

Method Code-Comment Inconsistency. Listing 2 shows a method of the `lenya` (APACHE..., 2022) project. This method iterates through an array of items of type `Item` and casts that type to the type `Group`. Each `Group` element is added to an array named `groups`. Finally, the method returns the `groups` array. Note that the comment "*Return an Iterator*" is inconsistent with what the method does. A better comment for that method would be "*Return an array of groups.*"

Techniques for checking code against a natural language description have been proposed, focusing mostly on cross-checking method-level comments against method implementations (PANTHAPLACKEL et al., 2021; TAN et al., 2012; RATOL; ROBILLARD, 2017; RABBI; SIDDIK, 2020). These techniques roughly fall into three groups. First, pattern-based techniques (TAN et al., 2012; RATOL; ROBILLARD, 2017) search for instances of pre-defined inconsistency patterns; they are powerful but do not generalize well to other kinds of inconsistencies. Second, there are learning-based techniques trained on manually labeled datasets (WANG et al., 2019; RABBI; SIDDIK, 2020). Due to inherent limits of manual labeling, such datasets are too small to fully exploit the power of deep learning models. Finally, models can be trained exclusively on automatically gathered datasets (PANTHAPLACKEL et al., 2021), which are large-scale, but risk including noisy and unrealistic examples, reducing the effectiveness of the resulting model.

1.1 PROPOSAL

This dissertation proposes TCID, a neural network-based approach for detecting text-code inconsistencies by learning a classifier that distinguishes consistent from inconsistent text-code pairs. The key challenge is how to obtain large amounts of adequate training data. In contrast to prior work, which relies either on manually labeled data alone (WANG et al., 2019; RABBI;

SIDDIK, 2020) or on automatically gathered data alone (PANTHAPLACKEL et al., 2021), TCID combines two complementary training strategies. At first, TCID performs weakly supervised[1] pre-training on a large-scale dataset of hundreds of thousands of automatically generated examples of consistent and inconsistent text-code pairs. TCID assigns "consistent" labels to the pairs that originate from open-source code as-is, and it assigns "inconsistent" labels to pairs obtained via random re-combination of text and code. Then, TCID fine-tunes the model on a small-scale dataset of manually labeled examples. These examples are curated to maximize the chance to fix high-confidence-but-wrong predictions during the fine-tuning, e.g., by sampling examples that the initial model is particularly certain about. Training models with this two-step procedure combines the benefits of (1) automatically gathered training data (a robust model able to generalize across diverse code examples) and of (2) human-provided (realistic) labels, while imposing a very moderate labeling effort. The two-step training procedure of TCID can be applied to arbitrary neural models and to different kinds of text-vs-code problems. In terms of models, we apply the approach to four state-of-the-art models: off-the-shelf LSTM and transformer models, and two models used in prior work (PANTHAPLACKEL et al., 2021) (a CodeBERT-based bag-of-words model and a GRU model). In terms of text-vs-code problems, we apply TCID to two problems and to datasets from two programming languages. One problem is to detect method-level Javadoc comments that are inconsistent with the method implementation. As a problem addressed in prior work (PANTHAPLACKEL et al., 2021), it allows for a direct comparison with the state-of-the-art. The other problem is to detect test descriptions in JavaScript that are inconsistent with the corresponding test cases. To the best of our knowledge, test-description inconsistencies have not been considered by any prior work, showing that TCID not only improves the state-of-the-art but also addresses a novel, practical problem.

1.2 RESULTS

Our evaluation applies the approach to 40,688 method-level comments from 1,518 Java projects and 338,761 test-description pairs gathered from 20,543 JavaScript projects. We find that a model trained only in a weakly supervised manner, i.e., on automatically generated and labeled examples, fails to perform well on a manually curated dataset (F_1=67% and F_1=57%). Fine-tuning the model on a small set of only 300 (for Java methods) and 194

[1] *Weak supervision* refers to supervised training with noisy labels (ZHOU, 2018).

(for JavaScript tests) manually labeled pairs, however, significantly improves the prediction accuracy (F_1=72% and F_1=74%). In particular, we show that our two-step training is beneficial across different models, and improves the effectiveness of state-of-the-art models for code-comment inconsistency detection (PANTHAPLACKEL et al., 2021). On the same dataset as previous work (PANTHAPLACKEL et al., 2021), our model improves the previously reported state-of-the-art accuracy from an F_1-score of 0.68 to 0.72. Finally, we validate the usefulness of TCID's predictions by submitting 34 pull requests, of which 10 have been accepted by the project owners so far.

1.3 OUTLINE

The remainder of this document is organized as follows:

Chapter 2 motivates the investigation of test-description inconsistencies. Chapter 3 explains how TCID works and how it was built. Chapter 4 evaluates TCID based on research questions. Chapter 5 presents the threats to validity of this work and Chapter 6 discusses related work. Finally, Chapter 7 concludes this dissertation.

2 STUDY ON TEST-VS-DESCRIPTION INCONSISTENCIES

This chapter reports on a preliminary study about the importance of detecting inconsistencies between test descriptions and corresponding test code. We investigate how common it is for developers to update test descriptions of JavaScript test cases. The rationale is that we are the first to investigate test-description inconsistencies, whereas the importance of detecting method-comment inconsistencies has already been established (TAN et al., 2007; TAN; YUAN; ZHOU, 2007; TAN et al., 2012).

2.1 PREVALENCE OF DESCRIPTION-ONLY UPDATES

This section addresses the question: *How common are updates of test descriptions only, i.e, updates that do not change the corresponding test code?* Answering this question gives an estimate of how commonly developers improve test descriptions.

To answer this question, we manually annotate a set of commits to test files. At first, we randomly mine popular JavaScript projects from GitHub that use the Jest test framework until we find 10 projects that each contain at least 20 commits updating Jest files. Then, we choose the 20 most recent commits updating Jest files from these 10 projects. The resulting 200 commits are split evenly between three of the authors, such that each commit gets annotated by two authors to answer the question: "Is there at least one test case where only the description was changed?". Cohen's kappa agreement among the three pairs of annotators ranges from 0.63 to 1.00, and after resolving disagreements, we find that 13 out of 200 commits (i.e., roughly one in every 15 commits) contain at least one instance of a description-only change.

Listing 3 – Example of description-only change

```
describe(
-    'Run jest test runner in watch mode. Passes through all flags directly to Jest'
+    'Run jest test runner. Passes through all flags directly to Jest'
  )
  .action(async () => {
    // Do this as the first thing so that any code reading it knows the right env.
    process.env.BABEL_ENV = 'test';
    process.env.NODE_ENV = 'test';
    // Makes the script crash on unhandled rejections instead of silently
    // ignoring them. In the future, promise rejections that are not handled will
    // terminate the Node.js process with a non-zero exit code.
    process.on('unhandledRejection', err => {
      throw err;
    });

    const argv = process.argv.slice(2);
    let jestConfig = {
      ...createJestConfig(
        relativePath => path.resolve(__dirname, '..', relativePath),
        paths.appRoot
      ),
      ...appPackageJson.jest,
    };
    try {
      // Allow overriding with jest.config
      const jestConfigContents = require(paths.jestConfig);
      jestConfig = { ...jestConfig, ...jestConfigContents };
    } catch {}

    argv.push(
      '--config',
      JSON.stringify({
        ...jestConfig,
      })
    );

    const [, ...argsToPassToJestCli] = argv;
    jest.run(argsToPassToJestCli);
  });
```

Source: TSDX (2023)

Listing 4 – Removed code block

```
  if (!process.env.CI) {
    argv.push('--watch'); // run jest in watch mode unless in CI
  }
```

Source: TSDX (2023)

Listing 3 shows an example of a description-only change that we found from the project

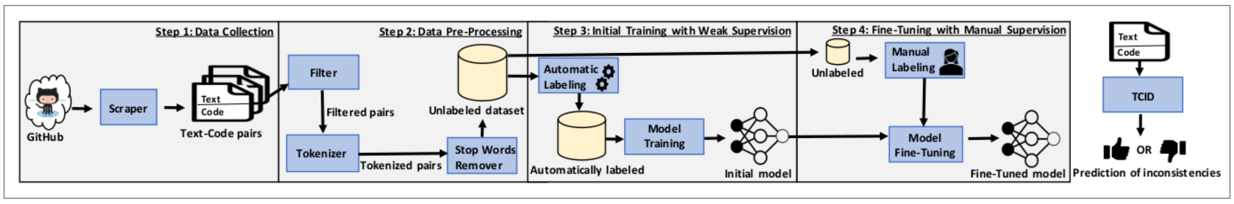
TSdx (TSDX, 2023). Notice that the description was changed from "*Run jest test runner in watch mode. Passes through all flags directly to Jest*" to "*Run jest test runner. Passes through all flags directly to Jest*". The previous description, "*Run jest test runner in watch mode. Passes through all flags directly to Jest*", specifies that the Jest test runner is ran in watch mode. However, the watch mode is not declared on the arguments passed to the Jest test runner. A closer look at the test file reveals that the watch mode was previously passed as argument to the Jest test runner and that during code evolution the code block presented in Listing 4 was removed from the test and the description was not modified to reflect the new behaviour of the test. In the commit where only the description is modified to "*Run jest test runner. Passes through all flags directly to Jest*", the commit message states that the test description was modified to properly reflect the behaviour of the test.

Given the total number of commits to Jest files on GitHub (more than 400,000 according to a search (SEARCH..., 2022)), and the fact that our annotations provide an underestimate of the prevalence of description-only updates, we conclude that developers commonly update test descriptions.

3 TCID: TEXT-CODE INCONSISTENCY DETECTION

This chapter describes TCID, our approach to detect text-code inconsistencies. TCID takes a pair of text and code as input and predicts the probability of them being inconsistent.

Figure 1 – Overview of the four-step workflow of TCID.



Source: Prepared by the author (2022)

Figure 1 illustrates the input and output of TCID at the far right. The large gray-colored rectangle in the figure shows the four-step workflow of TCID. In the first step, TCID mines code artifacts (e.g., methods or tests) from GitHub, parses the code, and extracts *text-code pairs* (Section 3.1). In the second step, TCID cleans and pre-processes the previously extracted pairs. The output of this step is a dataset of unlabeled pairs (Section 3.2). In the third step, TCID produces an initial model using weakly supervised learning, based on labels that a heuristic automatically produces for a given pair (Section 3.3). In more detail, TCID selects a large subset of the unlabeled data produced in the previous step, automatically labels the data, and then trains the initial neural model in a supervised manner on the automatically labeled text-code pairs. In the fourth and final step, TCID fine tunes the model using manual supervision (Section 3.4). To this end, human annotators label a small subset of pairs from the unlabeled dataset that have *not* been used to train the initial model.

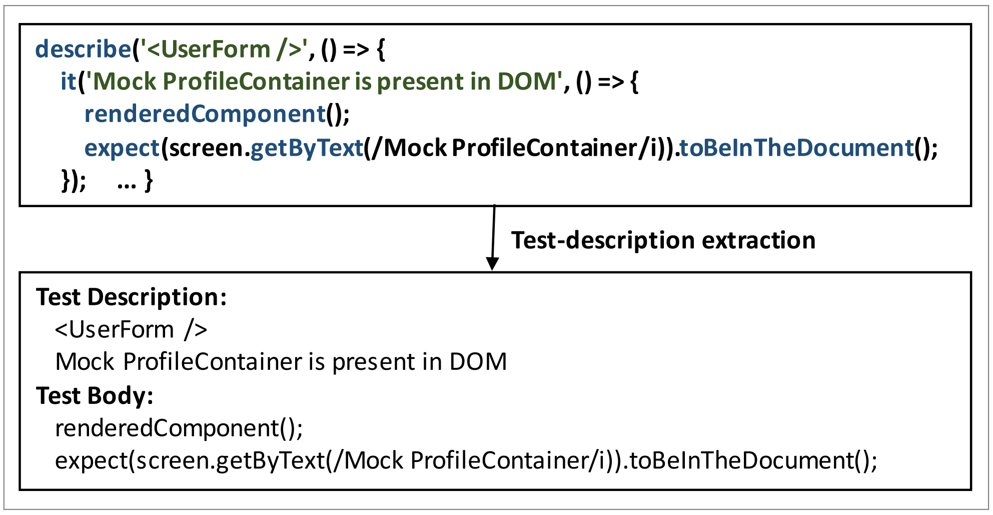
3.1 STEP 1: DATA COLLECTION

Given a corpus of code artifacts (e.g., methods or tests), TCID extracts *text-code pairs*.

Definition 1 (Text-code pair) *A text-code pair (t, c) consists of a textual description t associated with a code artifact c.*

To gather such pairs, we mine JavaScript projects from GitHub that use Jest as the test framework. For each repository, we run a lightweight static analysis to extract pairs of test case descriptions and code. The analysis searches for files that use the APIs for defining a test case and then parses each file into an abstract syntax tree (AST). Jest primarily uses three APIs to associate descriptions with test cases: `describe()`, which defines a block of several related tests, as well as `test()` and `it()` to define a single test. TCID identifies calls to these APIs in the AST and then extracts the relevant string literals. To form the test description, we merge the text associated with the `describe()` block and the `it()` (or `test()`) block of the test. Likewise, to form the test code, we merge the code from the following blocks: `beforeEach()` (or `beforeAll()`), `it()` (or `test()`), and `AfterEach()` (or `AfterAll()`). Figure 2 shows an example text-code pair. We mine 354,616 of such pairs distributed across 20,543 projects.

Figure 2 – Text-code pair for project `track` (TRACK..., 2021).



```
describe('<UserForm />', () => {
  it('Mock ProfileContainer is present in DOM', () => {
    renderedComponent();
    expect(screen.getByText(/Mock ProfileContainer/i)).toBeInTheDocument();
  });    ... }
```

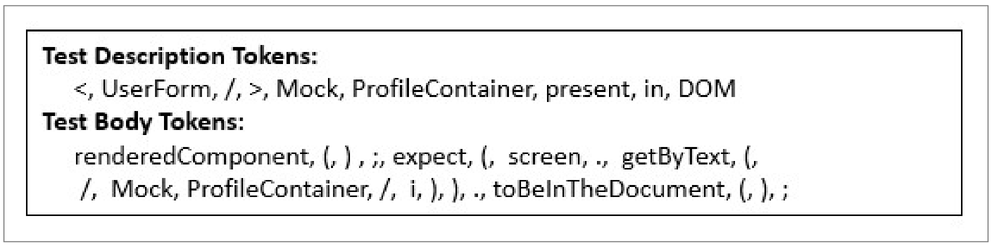
Test Description:
<UserForm />
Mock ProfileContainer is present in DOM
Test Body:
renderedComponent();
expect(screen.getByText(/Mock ProfileContainer/i)).toBeInTheDocument();

Source: Prepared by the author (2022)

3.2 STEP 2: DATA PRE-PROCESSING

To reduce noise in the dataset, we filter and preprocess the text-code pairs mined in Step 1. First, we discard pairs with empty descriptions, as those are not useful for checking consistency. Second, we discard pairs with a description in a language other than English, which we check using the FastText language model (BOJANOWSKI et al., 2017). FastText provides models for language identification, which can recognize 176 languages. We use the faster, slightly more accurate, and larger model provided (126 MB). Through experimentation, we noticed that some descriptions written in English are misidentified by this model due to camel case identifiers (e.g. 'get text' would be identified as English, and 'getText' would be identified as non-English but with low probability). Therefore, we keep descriptions identified as English or non-English with probability ≤ 0.15. After this process, we end up with a total of 338,761 pairs. Third, we tokenize the text and the code of each pair. Finally, we remove stop words appearing in the lists of tokens using the NLTK (NATURAL..., 2021) list of English stop words. Figure 3 shows the lists of pre-processed tokens for the text-code pair presented on Figure 2.

Figure 3 – List of tokens for the text-code pair from Figure 2.



Test Description Tokens:
<, UserForm, /, >, Mock, ProfileContainer, present, in, DOM
Test Body Tokens:
renderedComponent, (,) , ;, expect, (, screen, ., getByText, (,
/, Mock, ProfileContainer, /, i,),), ., toBeInTheDocument, (,), ;

Source: Prepared by the author (2022)

3.3 STEP 3: INITIAL TRAINING WITH WEAK SUPERVISION

Next, we describe the initial training step of TCID, which trains a neural model on an automatically labeled dataset. Section 3.3.1 describes the heuristic that we use to label pairs, and Section 3.3.2 elaborates on training the models.

3.3.1 Automatic Labeling

Deep neural models have been shown to be effective for reasoning about natural language and its relation to code (ALLAMANIS et al., 2018; PRADEL; CHANDRA, 2022). However, training an effective model in a supervised manner requires large amounts of labeled data, much more than can realistically be created via manual labeling. To obtain a large-scale labeled dataset of test-description pairs, we use 259,309 of the 338,761 (about 75%) pre-processed test-description pairs as a basis for labeling consistent and inconsistent examples.

The labeling procedure we use is as follows. All pairs mined from GitHub are kept as-is and receive the label 1, indicating that the test description and code are *consistent*.

To create pairs with the label 0, i.e., *inconsistent* test-description pairs, we proceed as follows. For each consistent pair, we create a copy of the pair and replace its description with the description of some other pair in the dataset. The rationale is that a random combination of a test and a description very likely results in an inconsistent pair.

The resulting dataset obtained with this procedure has 518,618 (=2*259,309) entries. The dataset is balanced by construction: (1) it contains the same number of positive and negative examples and (2) each test body is represented twice (in a positive example and in a negative example). We call this dataset the *automatically labeled dataset*.

3.3.1.1 Alternative Automatic Labeling Strategies

Beyond the automatic labeling described in Section 3.3.1, we consider three other labeling heuristics to address the mismatch between the automatically labeled training dataset and the more realistic, manually labeled test dataset. (1) We create inconsistent pairs by combining test code with descriptions of other test cases from the same file. The intuition is to create inconsistent pairs with more subtle changes relative to the base test description and code, so that the model can improve its discriminative power. As in the default automatic labeling, we consider all examples from GitHub as consistent. (2) In another attempt, for a given description-test pair (d, t), we use the description embedding model, which is pre-trained on all the test descriptions, to find a description d' that was most similar to d (according to the cosine similarity) and then create a negative pair (d', t) from it. Again, we consider all examples from GitHub as consistent. (3) Lastly, instead of considering all pairs mined from GitHub as consistent, we discard pairs with descriptions that are likely generic. Specifically, we manually create a list of words that are often used in generic test descriptions: “work”, “success”, “test”, “filter”, “failure”, “explodes”, “case”, “scenario”, and “screen”. Then, we use the description embedding model to assess whether a given description has a similar embedding to one of these words, and if similarity is above 0.5, then we discard the pair. For each positive pair that remains, we create a negative pair by randomly combining the body of the test with a random description or by combining the body with one of the words above.

Unfortunately, all of these heuristics yield models that perform poorly, and none of them outperforms the default automatic labeling described in Section 3.3.1, which randomly combines test descriptions and test code to produce inconsistent pairs. We conclude from these experiments that fully automatically creating a dataset that yields an effective model is hard, which provides empirical motivation for the two-step labeling strategy of TCID.

3.3.2 Model Training

In the following, we describe the neural models TCID uses and how they are trained with the automatically labeled dataset to build the *initial model*. TCID supports four models, all sharing the same overall structure: at first, they encode the given text and code into summary vectors, and then, they predict the probability that the two are consistent. The models are as follows:

- **LSTM.** The LSTM model uses two pre-trained FastText (BOJANOWSKI et al., 2017) models to embed the tokens of code and the text. FastText has been shown to be more effective on source code than other popular embeddings (WAINAKH; RAUF; PRADEL, 2021), such as Word2vec (MIKOLOV et al., 2013). The LSTM model then encodes the two resulting sequences of vectors into two summary vectors using a bi-directional recurrent neural network based on LSTM units (HOCHREITER; SCHMIDHUBER, 1997). Finally, the model predicts whether the text and code are consistent by feeding the summary vectors through one or more fully connected layers, which produce the classification result.
- **Transformer.** The Transformer model differs from the LSTM model by using a transformer-based module to encode the text and code (VASWANI et al., 2017). Transformers use multi-head attention to selectively focus on specific parts of the input and a positional encoding to communicate the order of the input vectors to the model.
- **CodeBERT BOW.** This model and the following are adapted from recent work (PANTHAPLACKEL et al., 2021) on identifying inconsistencies between a comment and code of a method. The CodeBERT BOW model computes the average CodeBERT (FENG et al., 2020) embedding vectors of the text and the code,[1] concatenates them, and feeds them through a feedforward neural network, which makes the final prediction.
- **SEQ.** The SEQ model encodes the text and the code with a Gated Recurrent Unit (GRU) (CHO et al., 2014) and then reasons about the relationship between the two by computing multi-head attention between each hidden state of the text encoder and the hidden states of the code encoder. The context vectors resulting from both attention mo-

[1] Consider the following embedding of a test, including only two tokens, for simplicity: $[[a_1, \ldots, a_m], [b_1, \ldots, b_m]]$. The BOW embedding for this test is $[(a_1 + b_1)/2, \ldots, (a_m + b_m)/2]$. The BOW embedding makes the token order irrelevant as the resulting embedding contains a single element.

Table 1 – Range of hyper-parameters considered during optimization.

Parameter	Range
Learning rate	1e-2, **1e-3**, 1e-4
Max. num. of epochs	**10**, 15, 20
Mini-batch size	**100**, 200
Max. num. of tokens in test descriptions	5, **10**, 15, 20
Max. num. of tokens in test bodies	10, 30, 50, **60**, 70, 100
Num. of layers of description encoder	1, **2**, 3
Num. of hidden layers of description encoder	**50**, 100, 150
Num. of hidden layers of test encoder	50, **100**, 150
Joint size of linear layer	100, **200**
Num. of layers of test encoder	1, **2**, 3
Dimension of the feedforward test encoder layer	100, **200**
Num. of heads of the test encoder layer	**2**, 3, 4

dules are then passed through another GRU encoder. The final state of this encoder is fed through fully-connected and softmax layers to obtain the final prediction.

Training Procedure. We use 80% of the data for training and 20% of the data for testing, respecting the distributions of classes within each partition, i.e., 50% consistent and 50% inconsistent. As the results obtained from untuned models can often be refuted, we followed the DSE/DUO approach (NAIR et al., 2018; AGRAWAL et al., 2020) and evaluated different hyperparameters of the model, as follows. We consider the parameters and range of values presented in Table 1 for the LSTM and transformer models. The highlighted rows are specific to the transformer model; the other rows are common to LSTM and transformer. To optimize model accuracy, we search for configurations (i.e., combinations of parameter assignments such as [learning rate=1e-2, max. num. of epochs=10, ...]) in the configuration space defined by the these parameters and ranges. It takes ∼25 min to train each model on a given configuration. Because of the high combinatorial cost of exploring all configurations, we randomly sample 30 configurations. Table 1 highlights in bold face the best configurations for the LSTM and transformer models. For the CodeBERT BOW and SEQ models, we use their default hyperparameters (PANTHAPLACKEL et al., 2021), which were already tuned. Chapter 4 reports results on those configurations.

3.4 STEP 4: FINE-TUNING WITH MANUAL SUPERVISION

The automatically labeled dataset suffers limitations, as many of the negative examples are easy to identify as inconsistent. The reason is that randomly recombining text and code often leads to pairs that obviously do not match. To address these inherent limitations of automatically labeling, we fine tune the model with a small set of manually annotated examples.

3.4.1 Selecting Pairs for Manual Labeling

Because manual labeling is expensive, we aim at maximizing the impact that fine-tuning on a small set of manually labeled examples will have. To this end, we select for manual labeling those examples for which the initial model provides high-confidence predictions, i.e., which, if proven to be wrong by a manual label, will cause the model to improve the most. Considering the task of detecting inconsistent test-description pairs, first, we select pairs from the 79,452 remaining pairs from the 338,761 pre-processed pairs. These pairs are distributed across 435 projects not overlapping with the ones on the automatically labeled dataset. Second, we sort the 79,452 pairs according to the predictions that the LSTM model of TCID produces when trained on the automatically labeled dataset. Third, we select 225 pairs from the top of the ranked-list of pairs and 225 pairs from the bottom of the list. The output of this step is a set of 450 pairs to label.

3.4.2 Manual Labeling

Each of the 450 pairs is labeled by two developers, enabling us to discard the cases where consensus cannot be reached. This task requires a total of 900 labels (as each pair requires two labels). To create an assignment to developers, we use a pairwise experimental design (MONTGOMERY, 2008), where each example is assigned to two developers. We recruited 12 developers to label text-code pairs according to their availability: half of the developers labeled 100 pairs (=6*100 labels) and the other half labeled 50 pairs (=6*50 labels). When assigning pairs to developers, we ensure that everybody receives an equal number of likely consistent and likely inconsistent pairs and, to reduce the risk of guessing labels, we shuffle the pairs before assigning them to developers.

To help developers with the labeling task, we prepare a guideline on how to execute the task, including examples. Each developer receives a separate spreadsheet containing the list of pairs (s)he needs to label. The spreadsheet contains a link to the test file on GitHub and the name of the test case declared within that file. Participants are expected to answer "Yes" or "No" to the question: *Does the test description explain the intent of the test code?* The rationale for using a Yes/No answer as opposed to a level scale is to reduce imprecision. Intuitively, the higher the number of options, the higher the chances of disagreement that could emerge from the different criteria used by humans during grading tasks.

At the end of the labeling task and after discarding the cases where participants disagree on the labels, we reach a total of 97 inconsistent pairs and 208 consistent pairs. There was disagreement in 145 (=450-(97+208)) of the 450 pairs (∼32% of the total) and we discard them. Furthermore, the ratio of consistent pairs is higher in this case compared to the ratio in the automatically labeled dataset (68% versus 50%).

As we aim at a balanced dataset, we randomly select 97 consistent pairs (of the 208 ones) to match the 97 inconsistent pairs. Thus, the small dataset consists of 194 examples, evenly distributed between consistent and inconsistent classes. We call the result of this labeling process the *manually labeled dataset*.

3.4.3 Model Fine-Tuning

TCID fine tunes the initial, already pre-trained model with the manually labeled dataset. This step allows the model to benefit both from the large amount of training data we obtain automatically and from the higher quality labels we obtain manually. Our evaluation shows that combining pre-training and fine-tuning is beneficial.

We consider the same set of parameters described in Section 3.3.2 and the same set of values assigned to them, except for *mini-batch size*, for which we reduce the list of options from [100, 200] to [10, 30, 50] because of the small size of the new set of examples. It takes ∼5 minutes to retrain each model on a given configuration in our setup. As in the initial training phase (Section 3.3), we randomly sample 30 configurations from our configuration space for the hyper-parameter optimization. The optimal configurations are the same as those reported in Section 3.3.2, except for mini-batches=10 and learning rate=1e-2. The results we report are based on these configurations.

4 EVALUATION

This Chapter presents the evaluation of TCID.

4.1 RESEARCH QUESTIONS

To better structure our evaluation, we rely on the Goal, Question, Metrics methodology (BASILI; CALDIERA; ROMBACH, 1994). The goal of our experiment consists of analyzing the effectiveness and usefulness of our approach to identify text-code inconsistencies when considering models trained with the two step training approach and in a weakly supervised manner only.

To achieve this goal, we address the following research questions:

RQ1. *How effective is* TCID *at identifying text-code inconsistencies?*

RQ2. *How does* TCID *compare to training state-of-the-art models only in a weakly supervised manner, i.e., without fine-tuning with a small amount of human-labeled data?*

RQ3. *Are the inconsistencies detected by* TCID *useful to software developers?*

4.2 EXPERIMENTAL SETUP

4.2.1 Metrics

We use standard metrics to evaluate our models, discussed below.

Accuracy. This metric indicates the number of correct predictions the model makes overall. The higher the value of this metric the better.

As Panthaplackel et al. (PANTHAPLACKEL et al., 2021), we measure precision, recall, and F1 with respect to the "Inconsistent" class. Intuitively, this class is important to the developer who looks to repair inconsistencies as opposed to confirm consistencies. More precisely, we report the following metrics.

Precision. This metric measures the number of predictions of inconsistency made by the model that are correct, i.e., that match the ground-truth label. The higher the value of this metric the better. High values indicate that the model should be trusted when it classifies a test-description pair as inconsistent.

Recall. This metric measures the number of examples with the label 0 (for "Inconsistent") that the model correctly predicted. The higher the value of this metric the better. High values indicate that the model missed few cases of inconsistent test-description pairs. More precisely, Recall measures the relative number of examples with label 0 that are correctly classified by the model.

F1. As usual, precision is a proxy for soundness whereas recall is a proxy for completeness. The F1-measure (DERCZYNSKI, 2016) considers these two complementary metrics together. It is the harmonic mean of precision and recall, given by $2 \times (p \times r)/(p + r)$, where p and r denote precision and recall, respectively.

4.2.2 Baseline

The closest existing work is by Panthaplackel et al. (PANTHAPLACKEL et al., 2021). Their main contribution is an approach for just-in-time text-code inconsistency detection that analyzes commits to determine whether they introduce an inconsistency into code assumed to be free of inconsistencies before the commit. We here consider a different, stationary usage scenario, which checks a code base in its current state for inconsistencies. A benefit of this stationary scenario is that it does not assume that the code base is consistent at some point in time. Panthaplackel et al. also present models, called "post hoc" (PANTHAPLACKEL et al., 2021), to be used in the stationary usage scenario. Their models are trained in a weakly supervised manner only. We consider these models, called CodeBERT BOW and SEQ, along with the weakly supervised training strategy as a baseline to compare TCID against the state-of-the-art.

4.2.3 Implementation and Hardware

Our implementation is based on PyTorch (PYTORCH, 2021), and we run all experiments on an NVIDIA Tesla P100 PCIe 16GB GPU.

Table 2 – Effectiveness of TCID (left) and weakly supervised pre-training only (right).

Model	**Supervision**							
	Weak + Manual				Weak Only			
	Inconsistency Type: *Method-vs-comment (Java)*							
	Acc.	**Pr.**	**Rec.**	**F1**	**Acc.**	**Pr.**	**Rec.**	**F1**
LSTM	0.55	0.56	0.67	0.55	0.59	0.62	0.48	0.54
Transformer	0.56	0.61	0.77	0.62	0.58	0.61	0.44	0.51
CodeBERT BOW	**0.73**	**0.73**	**0.72**	**0.72**	0.67	0.68	0.66	0.67
SEQ	0.63	0.68	0.54	0.60	0.57	0.56	0.66	0.61
	Inconsistency Type: *Test-vs-comment (JavaScript)*							
	Acc.	**Pr.**	**Rec.**	**F1**	**Acc.**	**Pr.**	**Rec.**	**F1**
LSTM	**0.73**	**0.75**	**0.72**	**0.74**	0.63	0.69	0.49	0.57
Transformer	0.70	0.74	0.68	0.71	0.66	0.72	0.51	0.59
CodeBERT BOW	0.68	0.68	0.68	0.67	0.62	0.77	0.35	0.47
SEQ	0.60	0.60	0.72	0.64	0.52	0.51	0.81	0.63

4.2.4 RQ1: Effectiveness of TCID

This section addresses the RQ "How effective is TCID at identifying text-code inconsistencies?". We evaluate the effectiveness of TCID at detecting method-vs-comment inconsistencies and on test-vs-description inconsistencies.

4.2.4.1 Method-vs-comment Inconsistencies

We use the dataset provided by previous work (PANTHAPLACKEL et al., 2021) on identifying inconsistencies between a comment and the code of a method. In total, this dataset contains 40,688 method-comment pairs from 1,518 Java projects. These pairs are automatically labeled based on commit histories of open-source Java projects (PANTHAPLACKEL et al., 2021). 300 of these pairs, from the testing partition of the full dataset, are manually labeled by humans. As for the datasets introduced in Sections 3.3.1 3.4, 50% of the pairs are consistent and 50% are inconsistent.

Initially, we train the models TCID uses on 32,988 automatically labeled pairs (the training partition of the dataset as provided by previous work (PANTHAPLACKEL et al., 2021)). Then, given the small amount of human-labeled data, i.e., 300 pairs, we fine-tune and evaluate the models using k-fold (k=10) stratified cross validation (BURMAN, 1989). That is, we partition the human-labeled dataset into k folds, each fold containing 300/k samples and repeat the following experiment for every one of the k folds. We select one fold for testing, train the model on the $k-1$ remaining folds, and evaluate the model on the initially-selected testing fold. Results are then averaged across the k experiments.

Table 2 shows the results under the section "Inconsistency Type: Method-vs-comment". For this task, TCID with the CodeBERT BOW model obtained the best result overall achieving an F1 measure of 0.72.

4.2.4.2 Test-vs-description Inconsistencies

Initially, we train the models TCID uses on 80% of the automatically labeled dataset that we mine from GitHub (Section 3.3.1). Then, to fine-tune and evaluate the models, we use the manually labeled dataset described on Section 3.4. Given the small amount of human-labeled data, we use k-fold (k=10) stratified cross validation, as we do for the method-vs-comment inconsistency task (Section 4.2.4.1). The difference here is that each fold contains 194/k samples. Table 2 shows the results under the section "Inconsistency Type: Test-vs-description". For this task, TCID with the LSTM model obtained the best result overall achieving an F1 measure of 0.74.

4.2.4.3 Examples

Listing 5 and Listing 6 show examples of real-world text-code inconsistencies found by the fine-tuned models, but not by the models trained with weak supervision only.

Listing 5 shows a method-vs-comment inconsistency from the Android External Spongycastle (ANDROID..., 2022) project. The method receives a `Certificate` as argument, encodes it and passes the encoded certificate as an argument to the `ASN1InputStream` constructor. Finally, the `readObject` method from the `ASN1InputStream` is called. This method returns an `ASN1Object`, which is the return type of the `toASN1Object` method. The comment, however, says that the method "*Return[s] a DERObject containing the encoded certificate.*", which is inconsistent with what the method does. A consistent comment would be "*Return a ASN1Object containing the encoded certificate.*".

Listing 6 shows a test-vs-description inconsistency found in a unit test of the React-Redux-Jest Boilerplate project. This test case expects the log of the "*test*" message to be undefined.

However, the description "*should console log our msg*" does not reflect this behavior, but instead suggests that there is a message. A better description for this test case would be "*Expects log of the 'test' message to be undefined*".

The two examples, which the models fail to detect when being trained with weak supervision only, illustrate the benefits of fine-tuning the models with manual supervision. At a cursory look, both text-code pairs appear to be consistent: In the example in Listing 5, both the method name and the comment are about some `Object`. In the example in Listing 6, both the test and its description are about logging. Only a more thorough inspection shows the semantic mismatch between the text and the code.

Listing 5 – Method-vs-comment Inconsistency identified only by the fine-tuned models

```
/** Return a DERObject containing the encoded certificate . */
private ASN1Object toASN1Object(X509Certificate cert)
  throws CertificateEncodingException {
    try {
      return new ASN1InputStream(cert.getEncoded()).readObject();
    } catch (Exception e) {
      throw new CertificateEncodingException(\"Exception while encoding certificate : \" +
          e.toString());
    }
  }
}
```

Source: ANDROID... (2022)

Listing 6 – Test-vs-description Inconsistency identified only by the fine-tuned models

```
describe('Log', () => {
  it('should console log our msg', ()=>{
    expect(log('test')).toBe(undefined);
  }); ...})
```

Source: REACT... (2022)

*Summary of **RQ1:*** TCID effectively detects both method-vs-comment inconsistencies (F1=0.72) and test-vs-description inconsistencies (F1=0.74).

4.2.5 RQ2: Comparison with Weakly Supervised Training Only

This section focuses on the RQ "How does TCID compare to training state-of-the-art models only in a weakly supervised manner, i.e., without fine-tuning with a small amount of human-labeled data?". As in Section 4.2.4, we apply all models that TCID uses to method-vs-comment and test-vs-description inconsistency detection.

4.2.5.1 Method-vs-comment Inconsistencies

With weakly supervised training only, we train the models on the training partition of the automatically labeled Java dataset and evaluate them on the manually labeled Java dataset. Table 2 shows the results in the "Weakly supervised pre-training only" columns under the "Method-vs-comment" inconsistency type. Overall, the results indicate that the models trained only on a large dataset of automatically labeled data are clearly weaker than TCID. For example, the F1-score of the CodeBERT BOW model drops from 0.72 to 0.67, and for the Transformer model it drops from 0.62 to 0.51. In particular, this result shows that TCID outperforms the baseline approach by Panthaplackel et al. (PANTHAPLACKEL et al., 2021) on their dataset.

4.2.5.2 Test-vs-description Inconsistencies

We train the models TCID uses on 80% of the training partition of the automatically labeled JavaScript dataset and evaluate them on the manually labeled JavaScript dataset. Table 2 shows the results in the "Weakly supervised pre-training only" columns under the "Test-vs-description" inconsistency type. As for the method-vs-comment inconsistency detection task, the results indicate that the models trained only on a large dataset of automatically labeled data are clearly worse than TCID: F1=0.57 versus F1=0.74 for the LSTM model, F1=0.59 versus F1=0.71 for the Transformer model, F1=0.47 versus F1=0.67 for the CodeBERT BOW model, and F1=0.63 versus F1=0.64 for the SEQ model.

Summary of ***RQ2:*** State-of-the-art models trained only with weakly supervised training do not reach the same level of effectiveness as TCID, both for method-vs-comment inconsistencies (F1=0.67 versus F1=0.72) and for test-vs-description inconsistencies (F1=0.57 versus F1=0.74).

4.2.6 RQ3: Usefulness

This section focuses on the RQ "Are the inconsistencies detected by TCID useful to software developers?". Since we are the first to investigate inconsistencies between test descriptions and the corresponding test case implementations, we address this question by applying TCID to open-source projects not used during training, and by creating pull requests (PRs) suggesting to improve test descriptions that TCID found to be inconsistent.

To identify candidate test-description pairs to prepare PRs, we first rank the predictions TCID makes on the validation partition of the automatically labeled dataset. Recall that half of the examples in this dataset are consistent test-description pairs (labeled as consistent); the other half of the pairs consists of random combinations of tests and descriptions (labeled as inconsistent). To detect issues in the existing open-source projects, we select (from the first partition of examples from the automatically labeled dataset) the 500 pairs that TCID predicts with highest probability to be inconsistent. Then, we keep only pairs from different projects, which leaves us with a total of 392 pairs. Finally, we manually analyze 50 of these 392 pairs. From the 50 analyzed pairs, we consider 36 to be indeed inconsistent, which roughly matches the precision of TCID found in RQ1. From these 36 inconsistent pairs, we discard two pairs. For one of them, the project from which it was extracted has been archived at GitHub. For the other one, the test from which the pair was extracted has been removed in the meantime.

For each of the remaining 34 inconsistent pairs, we prepare a PR suggesting an improved test description, which we manually create based on our understanding of the test intent. To preserve anonymity, we use an anonymous GitHub account to submit the PRs, and to avoid being considered a bot or having the account blocked, we contact the project owner prior to the PR with a personalized email explaining that we are working on a research project. Out

Table 3 – Merged pull requests.

aalluinmar/vue-test-jest/pull/1
aldeste/CrossWindowAuthPrototype/pull/1
AlexandrPristupa/Today-I-Learned/pull/7
anderfilth/payment-service-provider/pull/17
Gapur/place-management/pull/3
GroceriStar/json-file-schema-validator/pull/161
JelenaStrbac/customer-manager/pull/10
kmiloarguello/product-page-jumbo/pull/2
ozovalihasan/track/pull/4
rosivaldo9/node-com-jest/pull/10

of the 34 created PRs, 10 PRs have been accepted so far by the developers (29.4%). For two out of the 10 accepted PRs, the developers left a "*Thank you*" comment on the PR, suggesting gratefulness for the contribution. The remaining 24 PRs are still to be reviewed by the developers. None of our PRs has been rejected. Table 3 shows the 10 accepted PRs. The changes proposed to the test description from the "Track" project correspond to the example given in Chapter 1.

Summary of ***RQ3:*** Out of 34 submitted PRs, 10 have been accepted so far by the developers, and none were rejected. This result suggests that developers are receptive to test-description inconsistencies, and that TCID can be a useful addition to developer's toolkit.

5 THREATS TO VALIDITY

This chapter presents a summary of the potential threats to validity of our study, including external and internal validity.

5.1 EXTERNAL VALIDITY

TCID is affected by two threats to external validity. First, because our preliminary study is based on small samples of commits to test files and test description changes, it is possible that this data is not representative of all updates to test descriptions. The fact that developers accept PRs improving test descriptions, provides additional support for our hypothesis that developers care about updating test descriptions and that the inconsistencies identified by TCID can be useful. Second, to select inconsistencies to create pull requests, we manually inspect 50 pairs from the 500 predictions that TCID predicts with highest confidence as inconsistent. Moreover, we created PRs only for the 34 of these 50 pairs that we manually confirmed as inconsistent and for which the test case and project are still publicly available at GitHub. This experiment may not fully capture the usefulness of TCID, and in the future, we plan to analyze more cases detected as inconsistent and create a larger number of PRs.

5.2 INTERNAL VALIDITY

There are three potential threats to internal validity, i.e., reasons why our experiment may be ill-designed. We select pairs for manual labeling based on the predictions that the LSTM model of TCID produces when trained on the automatically labeled dataset. Additionally, our automatically labeled dataset is labeled based on the assumption that pairs mined from GitHub are consistent. While convenient for data collection, this assumption does not always hold in practice. However, in the labeling task of our manually labeled dataset, we find that the ratio of consistent pairs mined from GitHub is higher than the ratio of inconsistent pairs (68% versus 32%), which indicates that our assumption is valid for most cases. Finally, our manually labeled dataset is labeled by humans and may suffer from subjectivity bias. To mitigate this threat, (1) the manual labeling of each pair is independently performed by two developers and (2) we discard all the pairs for which no consensus is reached. Our results show that our manually labeled dataset improves effectiveness across different models, i.e., not only the model used to select pairs of test descriptions to manually label.

6 RELATED WORK

In this chapter, we describe some related work. We organized it in two groups: Training strategies in machine learning and text-code inconsistency detection.

6.1 TRAINING STRATEGIES IN MACHINE LEARNING

Our combination of large-scale weak supervision and manual fine-tuning relates to other training strategies in machine learning. One related idea is the general combination of pre-training and fine-tuning, as popularized, e.g., by large-scale language models (DEVLIN et al., 2018; CHEN et al., 2021). For these models, the pre-training task usually differs from the fine-tuning task, e.g., by pre-training on a next-token-prediction task and then fine-tuning on some classification task. Instead, TCID trains on the same task during both training phases. Another related idea is active learning, where the model actively and iteratively requests a user to annotate specific examples. In contrast, TCID does not require any interaction and can benefit from a pre-existing set of manually labeled examples, as in the method-vs-comment part of our evaluation. Finally, our approach relates to anomaly detection via one-class classification, but we formulate the problem as a binary classification task.

6.2 TEXT-CODE INCONSISTENCY DETECTION

Tan et al. (TAN et al., 2007; TAN; YUAN; ZHOU, 2007; TAN et al., 2012) pioneered in the area of code-documentation consistency. Their iComment technique (TAN et al., 2007) uses natural language processing, machine learning, and program analysis to extract implicit program rules from source code comments. iComment then uses these rules to detect inconsistencies between source code and comments related to the use of locking mechanisms. Many researchers have followed in Tan et al.'s footsteps over the last decade. Tan et al.'s own follow-up work @tcomment (TAN et al., 2012) can test the consistency between Javadoc comments related to null values and exceptions with the behavior of the corresponding code through a two step process: inferring a set of likely properties for a method from its comment, and generating random tests for these methods to check the inferred properties. Although there is bulk of work detecting inconsistencies between source code and documentation (TREUDE; MIDDLE-

TON; ATAPATTU, 2020), we are the first to look for inconsistencies between test code and its documentation. Test code has unique properties such as the often linear Arrange-Act-Assert (AAA) structure (GARVIN, 2021).

Automated attempts at mitigating the inconsistencies include DocRef by Zhong and Su (ZHONG; SU, 2013), which combines natural language and code analysis techniques to detect API documentation errors by identifying mismatches between code names in natural language documentation and code. AdDoc by Dagenais and Robillard (DAGENAIS; ROBILLARD, 2014) automatically discovers documentation patterns, i.e., coherent sets of code elements that are documented together. They found these patterns through the recovery of implicit links, and report violations of the patterns as the code and the documentation evolve. Similarly aimed at identifying inconsistencies between code and documentation, Ratol and Robillard (RATOL; ROBILLARD, 2017) presented Fraco, a tool to detect source code comments that are fragile with respect to identifier renaming, taking into account the type of identifier, its morphology, the scope of the identifier, and the location of comments.

A line of work by Zhou et al. (ZHOU et al., 2017; ZHOU et al., 2018; ZHOU et al., 2019) focused on detecting and repairing defects in Java API documentation. Their first approach (ZHOU et al., 2017) detects defects in API documents by leveraging techniques from program comprehension and natural language processing, with a particular focus on directives related to parameter constraints and exception throwing declarations. The authors employed a first-order logic based constraint solver to detect such defects. Their follow-up work (ZHOU et al., 2018; ZHOU et al., 2019) introduced a solution to repair the detected defects, focusing on situations when the constraint description of API usage is incomplete, or when the description exists but is semantically incorrect with respect to the code.

Focusing on a particular type of source code comment, i.e., TODO comments, Sridhara (SRIDHARA, 2016) presented a technique that can automatically detect the status of a TODO comment by employing a combination of information retrieval, linguistics and semantics. Along similar lines, Maipradit et al. (MAIPRADIT et al., 2020) defined the concept of "on-hold" self-admitted technical debt, i.e., debt which contains a condition to indicate that a developer is waiting for a certain event or an updated functionality having been implemented elsewhere, followed by an approach to check if the "on-hold" instances are superfluous and can be removed, e.g., because the referenced issue has been closed (MAIPRADIT et al.,). Recent work by Gao et al. (GAO et al., 2021) follows in a similar direction and introduces an approach called TDCleaner to identify obsolete TODO comments by enabling just-in-time checking of the status of TODO comments.

Similar to the work by Gao et al. and our work, much of the recent work on detecting and mitigating inconsistencies between natural language documentation and source code has focused on employing machine learning and deep learning approaches. Panthaplackel et al. (PANTHAPLACKEL et al., 2021) proposed an approach that learns to correlate changes across two distinct language representations, to generate a sequence of edits that are applied to existing comments to reflect code modifications. Rabbi and Siddik (RABBI; SIDDIK, 2020) proposed an approach for resolving the inconsistencies between code and comments using a Siamese recurrent network, which uses word tokens in code and comments as well as their sequences in corresponding code or comments. In follow-up work, Rabbi et al. (RABBI et al., 2020) proposed an ensemble approach for detecting inconsistencies using topic modeling by automatically extracting dominant topics and identifying the consistency between a code snippet and its corresponding comment. Patra and Pradel (PATRA; PRADEL, 2022) propose to find inconsistencies between the name of a variable and the values it refers to. None of these approaches addresses the problem of test-description inconsistencies.

To enable just-in-time updating of comments when the corresponding code changes, Liu et al. (LIU et al., 2020) introduced Comment UPdater (CUP), a tool which leverages a neural sequence-to-sequence model to learn comment update patterns from existing code-comment co-changes and can automatically generate a new comment based on its corresponding old comment and code change. Lin et al. (LIN et al., 2021) found that the overall effectiveness of CUP is largely contributed by its success on updating comments via a single token change. From that, the authors proposed HEBCUP as a heuristic-based alternative to CUP, finding that it outperforms CUP.

To assist developers in writing natural language documentation that is less likely to suffer from inconsistencies, Nassif et al. (NASSIF et al., 2021) introduced an approach called DScribe which allows developers to combine unit tests and documentation templates, and to invoke those templates to generate documentation and unit tests. The tool supports the detection and replacement of outdated documentation, and the use of templates can encourage extensive test suites with a consistent style.

7 CONCLUSION AND FUTURE WORK

Source code is often accompanied by natural language text, e.g., in the form of source code comments or test code descriptions. Unfortunately, code and text are not always consistent, which can mislead developers during program comprehension tasks and can be indicative of faults. To support the automated detection of such inconsistencies, we present TCID, which trains a neural model that classifies text-code pairs as consistent or inconsistent. Since models trained only on automatically labeled data often suffer from low accuracy caused by many of the automatically generated data points not being representative of real-world mistakes, TCID employs a two-phase training strategy: At first, the model is trained using weak supervision on a large-scale dataset of automatically generated and labeled examples of text-code pairs. Then, the model is fine-tuned using manual supervision based on a small-scale dataset of manually labeled examples.

Our evaluation with 40,688 method-comment pairs in Java and 338,761 test-description pairs in JavaScript shows that a small amount of manual labeling can significantly improve the detection of inconsistencies. In particular, TCID outperforms the state-of-the-art on the method-vs-comment task, and is the first to address the test-vs-description task. To validate the usefulness of the approach, we prepared and submitted pull requests, of which 10 have been accepted so far.

7.1 CONTRIBUTIONS

- ⋆ A two-step training approach combining the benefits of weakly supervised pre-training on large-scale, automatically-gathered data with the benefits of fine-tuning on manually labeled data, which we find crucial for effective learning.
- ⋆ The first to address the problem of identifying inconsistencies between test descriptions and corresponding test case implementations.
- ⋆ Empirical evidence that the approach effectively detects inconsistent text-code pairs, outperforms the state-of-the-art, and provides value to developers.

7.2 FUTURE WORK

- ⋆ Investigate how to improve TCID's effectiveness by considering (1) other heuristics to automatically label consistent and inconsistent text-code pairs, (2) multiple fine-tuning iterations and (3) fine-tuning large pre-trained models, e.g. CodeT5 (WANG et al., 2021).
- ⋆ Evaluate TCID in other scenarios by considering (1) different text-code pairs such as requirement and implementation, (2) a larger number of PRs.

REFERENCES

AGRAWAL, A.; MENZIES, T.; MINKU, L. L.; WAGNER, M.; YU, Z. Better software analytics via "duo": Data mining algorithms using/used-by optimizers. *Empirical Software Engineering*, v. 25, n. 3, p. 2099–2136, 2020.

ALLAMANIS, M.; BARR, E. T.; DEVANBU, P.; SUTTON, C. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, ACM, v. 51, n. 4, p. 81, 2018.

ANDROID External Spongycastle. 2022. Available at: <https://github.com/byterom/android_external_spongycastle>.

APACHE Lenya. 2022. Available at: <https://github.com/apache/lenya>.

BASILI, V. R.; CALDIERA, G.; ROMBACH, H. D. The goal question metric approach. In: . [S.l.: s.n.], 1994.

BOJANOWSKI, P.; GRAVE, E.; JOULIN, A.; MIKOLOV, T. Enriching word vectors with subword information. *TACL*, v. 5, p. 135–146, 2017. Available at: <https://transacl.org/ojs/index.php/tacl/article/view/999>.

BURMAN, P. A comparative study of ordinary cross-validation, v-fold cross-validation and the repeated learning-testing methods. *Biometrika*, [Oxford University Press, Biometrika Trust], v. 76, n. 3, p. 503–514, 1989. ISSN 00063444. Available at: <http://www.jstor.org/stable/2336116>.

CHEN, M.; TWOREK, J.; JUN, H.; YUAN, Q.; PINTO, H. P. de O.; KAPLAN, J.; EDWARDS, H.; BURDA, Y.; JOSEPH, N.; BROCKMAN, G.; RAY, A.; PURI, R.; KRUEGER, G.; PETROV, M.; KHLAAF, H.; SASTRY, G.; MISHKIN, P.; CHAN, B.; GRAY, S.; RYDER, N.; PAVLOV, M.; POWER, A.; KAISER, L.; BAVARIAN, M.; WINTER, C.; TILLET, P.; SUCH, F. P.; CUMMINGS, D.; PLAPPERT, M.; CHANTZIS, F.; BARNES, E.; HERBERT-VOSS, A.; GUSS, W. H.; NICHOL, A.; PAINO, A.; TEZAK, N.; TANG, J.; BABUSCHKIN, I.; BALAJI, S.; JAIN, S.; SAUNDERS, W.; HESSE, C.; CARR, A. N.; LEIKE, J.; ACHIAM, J.; MISRA, V.; MORIKAWA, E.; RADFORD, A.; KNIGHT, M.; BRUNDAGE, M.; MURATI, M.; MAYER, K.; WELINDER, P.; MCGREW, B.; AMODEI, D.; MCCANDLISH, S.; SUTSKEVER, I.; ZAREMBA, W. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021. Available at: <https://arxiv.org/abs/2107.03374>.

CHO, K.; MERRIENBOER, B. van; BAHDANAU, D.; BENGIO, Y. *On the Properties of Neural Machine Translation: Encoder-Decoder Approaches*. 2014.

DAGENAIS, B.; ROBILLARD, M. P. Using traceability links to recommend adaptive changes for documentation evolution. *IEEE Transactions on Software Engineering*, v. 40, n. 11, p. 1126–1146, 2014.

DERCZYNSKI, L. Complementarity, F-score, and NLP evaluation. In: *Tenth International Conference on Language Resources and Evaluation (LREC)*. [S.l.]: European Language Resources Association (ELRA), 2016. p. 261–266.

DEVLIN, J.; CHANG, M.; LEE, K.; TOUTANOVA, K. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018. Available at: <http://arxiv.org/abs/1810.04805>.

ERNST, M. D. Natural language is a programming language: Applying natural language processing to software development. In: *SNAPL*. [S.l.: s.n.], 2017.

FENG, Z.; GUO, D.; TANG, D.; DUAN, N.; FENG, X.; GONG, M.; SHOU, L.; QIN, B.; LIU, T.; JIANG, D.; ZHOU, M. Codebert: A pre-trained model for programming and natural languages. In: COHN, T.; HE, Y.; LIU, Y. (Ed.). *Findings of the Association for Computational Linguistics (EMNLP)*. [S.l.]: Association for Computational Linguistics, 2020. (Findings of ACL, EMNLP 2020), p. 1536–1547.

GAO, Z.; XIA, X.; LO, D.; GRUNDY, J.; ZIMMERMANN, T. Automating the removal of obsolete todo comments. *arXiv preprint arXiv:2108.05846*, 2021.

GARVIN, T. M. *Principles of Test Driven Development*. 2021. Https://integralpath.blogs.com/thinkingoutloud/2005/09/principles$_of_t.html$.

HINDLE, A.; BARR, E. T.; SU, Z.; GABEL, M.; DEVANBU, P. On the naturalness of software. In: *34th International Conference on Software Engineering*. [S.l.]: IEEE, 2012. (ICSE), p. 837–847. ISBN 9781467310673.

HOCHREITER, S.; SCHMIDHUBER, J. Long short-term memory. *Neural computation*, MIT Press, v. 9, n. 8, p. 1735–1780, 1997.

LIN, B.; WANG, S.; LIU, K.; MAO, X.; BISSYANDÉ, T. F. Automated comment update: How far are we? In: *IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. [S.l.]: IEEE, 2021. p. 36–46.

LIU, Z.; XIA, X.; YAN, M.; LI, S. Automating just-in-time comment updating. In: *35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.: s.n.], 2020. p. 585–597.

MAIPRADIT, R.; LIN, B.; NAGY, C.; BAVOTA, G.; LANZA, M.; HATA, H.; MATSUMOTO, K. Automated identification of on-hold self-admitted technical debt. In: *IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. [S.l.]: IEEE. p. 54–64.

MAIPRADIT, R.; TREUDE, C.; HATA, H.; MATSUMOTO, K. Wait for it: identifying "on-hold" self-admitted technical debt. *Empirical Software Engineering*, Springer, v. 25, n. 5, p. 3770–3798, 2020.

MIKOLOV, T.; SUTSKEVER, I.; CHEN, K.; CORRADO, G. S.; DEAN, J. Distributed representations of words and phrases and their compositionality. In: *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*. [S.l.: s.n.], 2013. p. 3111–3119.

MONTGOMERY, D. *Design and Analysis of Experiments*. [S.l.]: John Wiley & Sons, 2008. (Student solutions manual). ISBN 9780470128664.

NAIR, V.; AGRAWAL, A.; CHEN, J.; FU, W.; MATHEW, G.; MENZIES, T.; MINKU, L.; WAGNER, M.; YU, Z. Data-driven search-based software engineering. In: *15th International Conference on Mining Software Repositories (MSR)*. [S.l.]: ACM, 2018. p. 341–352. ISBN 9781450357166.

NASSIF, M.; HERNANDEZ, A.; SRIDHARAN, A.; ROBILLARD, M. P. Generating unit tests for documentation. *IEEE Transactions on Software Engineering*, IEEE, 2021.

NATURAL Language Toolkit. 2021. Available at: <https://www.nltk.org/>.

PANTHAPLACKEL, S.; LI, J. J.; GLIGORIC, M.; MOONEY, R. J. 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*. [S.l.]: AAAI Press, 2021. p. 427–435.

PATRA, J.; PRADEL, M. Nalin: Learning from runtime behavior to find name-value inconsistencies in jupyter notebooks. In: *ICSE*. [S.l.: s.n.], 2022.

PRADEL, M.; CHANDRA, S. Neural software analysis. *Commun. ACM*, v. 65, n. 1, p. 86–96, 2022. Available at: <https://doi.org/10.1145/3460348>.

PYTORCH. 2021. Available at: <https://pytorch.org/>.

RABBI, F.; HAQUE, M. N.; KADIR, M. E.; SIDDIK, M. S.; KABIR, A. An ensemble approach to detect code comment inconsistencies using topic modeling. In: *32nd International Conference on Software Engineering and Knowledge Engineering (SEKE)*. [S.l.: s.n.], 2020. p. 392–395.

RABBI, F.; SIDDIK, M. S. Detecting code comment inconsistency using siamese recurrent network. In: *28th International Conference on Program Comprehension*. [S.l.]: ACM, 2020. (ICPC), p. 371–375. ISBN 9781450379588.

RATOL, I. K.; ROBILLARD, M. P. Detecting fragile comments. In: *32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.: s.n.], 2017. p. 112–122.

REACT, Redux, Jest Boilerplate Project. 2022. Available at: <https://github.com/JoeKarlsson/react-redux-boilerplate>.

SEARCH for commits with test description. 2022. <https://github.com/search?q=jest&type=Commits&ref=advsearch&l=&l=>.

SRIDHARA, G. Automatically detecting the up-to-date status of todo comments in java programs. In: *9th India Software Engineering Conference (ISEC)*. [S.l.: s.n.], 2016. p. 16–25.

TAN, L.; YUAN, D.; KRISHNA, G.; ZHOU, Y. /* icomment: Bugs or bad comments?*/. In: *21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. [S.l.: s.n.], 2007. p. 145–158.

TAN, L.; YUAN, D.; ZHOU, Y. Hotcomments: how to make program comments more useful? In: *HotOS*. [S.l.]: USENIX Association, 2007. v. 7, p. 49–54.

TAN, S. H.; MARINOV, D.; TAN, L.; LEAVENS, G. T. @ tcomment: Testing javadoc comments to detect comment-code inconsistencies. In: *IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*. [S.l.]: IEEE, 2012. p. 260–269.

TRACK It (Dr. Ti). 2021. Available at: <https://github.com/ozovalihasan/track/blob/development/src/component/UserForm/test/UserForm.test.js>.

TREUDE, C.; MIDDLETON, J.; ATAPATTU, T. Beyond accuracy: Assessing software documentation quality. In: *28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. [S.l.]: ACM, 2020. p. 1509–1512.

TSDX. 2023. Available at: <https://github.com/jaredpalmer/tsdx>.

VASWANI, A.; SHAZEER, N.; PARMAR, N.; USZKOREIT, J.; JONES, L.; GOMEZ, A. N.; KAISER, L.; POLOSUKHIN, I. Attention is all you need. In: *NIPS*. [s.n.], 2017. p. 6000–6010. Available at: <http://papers.nips.cc/paper/7181-attention-is-all-you-need>.

WAINAKH, Y.; RAUF, M.; PRADEL, M. 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. p. 562–573. Available at: <https://doi.org/10.1109/ICSE43902.2021.00059>.

WANG, D.; GUO, Y.; DONG, W.; WANG, Z.; LIU, H.; LI, S. Deep code-comment understanding and assessment. *IEEE Access*, v. 7, p. 174200–174209, 2019.

WANG, Y.; WANG, W.; JOTY, S. R.; HOI, S. C. H. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *ArXiv*, abs/2109.00859, 2021.

ZHONG, H.; SU, Z. Detecting api documentation errors. In: *International Conference on Object-oriented Programming Systems Languages, and Applications (OOPSLA)*. [S.l.]: ACM, 2013. p. 803–816.

ZHOU, Y.; GU, R.; CHEN, T.; HUANG, Z.; PANICHELLA, S.; GALL, H. Analyzing apis documentation and code to detect directive defects. In: *IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. [S.l.]: IEEE, 2017. p. 27–37.

ZHOU, Y.; WANG, C.; YAN, X.; CHEN, T.; PANICHELLA, S.; GALL, H. C. Automatic detection and repair recommendation of directive defects in java api documentation. *IEEE Transactions on Software Engineering*, v. 46, n. 9, p. 1004–1023, 2018.

ZHOU, Y.; YAN, X.; CHEN, T.; PANICHELLA, S.; GALL, H. Drone: a tool to detect and repair directive defects in java apis documentation. In: *International Conference on Software Engineering (ICSE): Companion Proceedings*. [S.l.]: IEEE, 2019. p. 115–118.

ZHOU, Z.-H. A brief introduction to weakly supervised learning. *National science review*, Oxford University Press, v. 5, n. 1, p. 44–53, 2018.