Facilitating Software Evolution through Natural Language Comments and Dialogue

Sheena Panthaplackel
The University of Texas at Austin
spantha@cs.utexas.edu

Doctoral Dissertation Proposal

Abstract

Software projects are continually evolving, as developers incorporate changes to refactor code, support new functionality, and fix bugs. To uphold software quality amidst constant changes and also facilitate prompt implementation of critical changes, it is desirable to have automated tools for guiding developers in making methodical software changes. We explore tasks and data and design machine learning approaches which leverage natural language to serve this purpose.

When developers make code changes, they sometimes fail to update the accompanying natural language comments documenting various aspects of the code, which can lead to confusion and vulnerability to bugs. We present our completed work on alerting developers of inconsistent comments upon code changes and suggesting updates by learning to correlate comments and code.

When a bug is reported, developers engage in a dialogue to collaboratively understand it and ultimately resolve it. While the solution is likely formulated within the discussion, it is often buried in a large amount of text, making it difficult to comprehend, which delays its implementation through the necessary repository changes. To guide developers in more easily absorbing information relevant towards making these changes and consequently expedite bug resolution, we investigate generating a concise natural language description of the solution by synthesizing relevant content as it emerges in the discussion. In completed work, we benchmark models for generating solution descriptions and design a classifier for determining when sufficient context for generating an informative description becomes available. We also investigate a pipelined approach for real-time generation, entailing separate classification and generation models.

For future work, we propose an improved classifier and also a more intricate system that is jointly trained on generation and classification. Next, we intend to study a system which can interactively generate natural language descriptions which can drive code changes. Finally, we plan to investigate how we can leverage the discussion context to also suggest concrete code changes for bug resolution.

Contents

Chapter 1

Introduction

Natural language serves as an important medium for search, documentation, and communication throughout the software development process. Developers search online code bases using natural language queries when they are trying to find a code implementation of a particular functionality. They write natural language comments alongside source code in order to document key aspects of the code. When a software bug is found, a user opens an issue report, in which developers engage in a natural language dialogue to collectively resolve the bug. To foster the role of natural language in software development, there is growing interest in building AI-driven tools for various tasks, such as code search and comment generation.

We present novel tasks, datasets, and machine learning approaches which leverage natural language to facilitate *software evolution*. Software projects are highly dynamic in nature, with developers continually incorporating changes for refactoring code, supporting new functionality, and fixing bugs. When projects are collectively developed across large teams and through agile development practices emphasizing software flexibility, the number of developers making changes and frequency of these changes increase drastically. Due to the sheer volume, there is a high risk for overall software quality to deteriorate as developers may unintentionally introduce potential vulnerabilities when they make changes. Moreover, from the large mass of changes that need to be made, those that are the most pressing (e.g., critical bug fixes) can easily get delayed, especially when developers are preoccupied by their present assignments or are less familiar with the relevant components of the project. We present ways in which natural language can be used to guide developers in making more methodical software changes.

For our first goal of upholding software quality upon code changes, we focus on natural language comments. Many code changes require reciprocal updates to the accompanying comments to keep them in sync; however, this is not always done in practice. Outdated comments which inaccurately portray the code they accompany adversely affect the software development cycle by causing confusion and misguiding developers, hence making code vulnerable to bugs. We present our work on *just-in-time* inconsistency detection (Chapter 4), for alerting developers of inconsistency immediately upon code changes. To help them revise comments to reflect these code changes, we investigate generating recommended comment revisions. For this, we design a framework which learns to correlate changes across two distinct language representations, to generate a sequence of edits that are applied to the existing comment to reflect the source code modifications (Chapter 5). We combine the detection and generation models to build a more comprehensive automatic comment maintenance system that detects and resolves inconsistencies

(Chapter 6). To relate code and comments for such cross-modal tasks, we employ a rich feature set derived from our work in learning explicit associations between entities in a comment and elements in the corresponding code (Chapter 3).

Next, to address the second goal of driving critical code changes, specifically changes for resolving bugs threatening software quality, we consider natural language dialogue in bug report discussions. Bug resolution is often strenuous and time-consuming, involving extended deliberations among multiple participants, spanning long periods of time. Although a solution often emerges within the bug report discussion, this can easily get lost in a large amount of text. Wading through a long discussion to determine whether a solution has been recommended, comprehending it, and then implementing it through the necessary code or documentation changes in the code base can be daunting, especially for developers who are not closely following the discussion. This delays implementation, and consequently, the bug persists in the code base, threatening the reliability of the software. As developers scan through the long discussion, it is desirable to have an automated system which can guide them to more easily absorb information relevant towards implementing the changes. To address this, we study generating a concise natural language description of the solution by synthesizing relevant content in the discussion (Chapter 7). To help quickly mobilize developers for implementation and expedite bug resolution in a real-time setting, the description should be generated as soon as the necessary context for generating an informative description emerges in the discussion. For this, we study a classification task for determining when this context becomes available and conduct develop a pipelined approach as an initial investigation for a real-time generation system (Chapter 8).

As our first short-term goal, we propose studying pretrained language models to develop a higher-performing classifier for determining when sufficient context emerges (Section 9.1). Next, since generating solution descriptions and classifying whether sufficient context for generating an informative description are interdependent tasks, our second short-term goal revolves around building a jointly trained system which allows the two tasks to complement one another (Section 9.2). This system is designed to generate a natural language description to guide a developer in implementing a single set of code changes; however, making code changes is often an iterative process. To support this process, our first long-term goal is building a system for interactively generating descriptions to guide code changes (Section 10.1). Finally, while a description can provide a high-level overview of the changes to be implemented, developers must still reason about how it should manifest as concrete code changes. To help developers with this, we propose building a system which leverages the discussion context to generate suggested code changes for bug resolution (Section 10.2).

Chapter 2

Background and Related Work

2.1. Natural Language + Source Code

There is growing interest in cross-modal tasks, combining various forms of natural language (NL) with source code. Code generation for a given NL input is a popular task (Dong and Lapata, 2016; Lin et al., 2018; Rabinovich et al., 2017; Yin and Neubig, 2017; Agashe et al., 2019; Shin et al., 2019; Ye et al., 2020; Sun et al., 2020; Xu et al., 2020; Wang et al., 2020a; Dahal et al., 2021). Husain et al. (2019), Cambronero et al. (2019), Zhao and Sun (2020), and Haldar et al. (2020) explore code search based on NL queries. Prior work examines tasks for generating natural language commit messages (Loyola et al., 2017; Xu et al., 2019a) and pull request (PR) descriptions (Liu et al., 2019) to characterize code changes.

There is also extensive work in generating NL descriptions of code. For this, Iyer et al. (2016), Yao et al. (2018), and Yin et al. (2018) consider StackOverflow question titles paired with corresponding code snippets in the answers. Allamanis et al. (2016), Xu et al. (2019b), Alon et al. (2019), and Fernandes et al. (2019) consider method names paired with method bodies. Sridhara et al. (2011), Sridhara et al. (2010), Movshovitz-Attias and Cohen (2013), Hu et al. (2018), Liang and Zhu (2018), LeClair et al. (2019), Fernandes et al. (2019), Ahmad et al. (2020), and Yu et al. (2020) consider comments paired with methods or classes.

2.2. Source Code Comments

Natural language comments appear alongside source code in the form of single-line comments, block comments, and documentation comments for classes and methods (Oracle, 2021). Comments document various aspects of code, including functionality, usage, implementation, and error cases (Pascarella and Bacchelli, 2017). Comments are critical for program readability (Tenny, 1988) and comprehension (Woodfield et al., 1981), and consequently, software maintenance (Oman and Hagemeister, 1992).

There have been some efforts to model granular associations between natural language and source code. Li and Boyer (2015, 2016) ground noun phrases within an educational dialogue system to a programming environment and Liu et al. (2018a) link different change intents contained in a single commit message to source code files in a software project which have changed within the commit. However, there is very limited work which studies such associations between *comments* and source code. While there is work that maps a single source code component (e.g., class, method, statement) to a comment based on distance metrics and other simple heuristics (Fluri et al., 2007), this does not capture the more fine-grained associations, which we study in Chapter 3.

2.3. Software Evolution

To quickly deliver software to users, software teams generally prioritize implementing the simplest solution to meet current needs rather than designing a more involved solution which anticipates future needs (Turk et al., 2005). Such a strategy requires a high degree of flexibility, as developers must be able to adapt the software when new requirements emerge in the future, for improving or extending existing functionality, enhancing performance, or making it compatible with new environments (Lehman and Fernáandez-Ramil, 2006). In addition to adding code for addressing these requirements, developers must also *refactor* existing code to be able to efficiently integrate the new code (Nyamawe et al., 2019). Efforts to resolve defects causing unintended behavior, or bugs (Murphy-Hill et al., 2015), also contribute to software evolution. Bugs form as a result of faulty code, invalid assumptions, or incompatibility to external dependencies (Rodríguez-Pérez et al., 2020).

Recently, there is growing work in modeling code changes for facilitating software evolution. Yin et al. (2019) and Hoang et al. (2020) aim to learn vector representations for common code change patterns, and Chakraborty et al. (2020) and Yao et al. (2021) focus on learning to apply common code edits. There have also been efforts to address more specialized forms of code editing, including bug fixing (Kim et al., 2013; Ke et al., 2015; Le et al., 2017, 2016; Le Goues et al., 2012; Tufano et al., 2019b), resolving compilation errors (Campbell et al., 2014; Gupta et al., 2017; Mesbah et al., 2019; Tarlow et al., 2020), refactoring (Tansey and Tilevich, 2008; Raychev et al., 2013; Ge et al., 2012; Meng et al., 2015), and suggesting API-related edits (Nguyen et al., 2010, 2016). Brody et al. (2020) and Foster et al. (2012) study the task of predicting edit completions for partially edited code snippets and Miltner et al. (2019) put forth edit suggestions by observing repetitive edits made by the user.

2.4. Comment/Code Inconsistency

As source code evolves, the accompanying comments must be updated accordingly; however, developers often fail to do this (Wen et al., 2019; Fluri et al., 2009; Ratol and Robillard, 2017; Jiang and Hassan, 2006; Zhou et al., 2017; Tan et al., 2007). Outdated comments lead to confusion (Wen et al., 2019; Jiang and Hassan, 2006; Tan et al., 2007; Zhou et al., 2017) and vulnerability to bugs (Jiang and Hassan, 2006; Tan et al., 2007; Ibrahim et al., 2012). Prior work analyze how inconsistencies emerge (Fluri et al., 2009; Jiang and Hassan, 2006; Ibrahim et al., 2012; Fluri et al., 2007) and the various types of inconsistencies (Wen et al., 2019).

To address this, prior work propose rule-based approaches for detecting pre-existing inconsistencies in specific domains, including locks (Tan et al., 2007), interrupts (Tan et al., 2011), `null` exceptions for method parameters (Zhou et al., 2017; Tan et al., 2012), and renamed identifiers (Ratol and Robillard, 2017). The comments they consider are consequently constrained to certain templates relevant to their respective domains. Corazza et al. (2018) and Cimasa et al. (2019) address a broader notion of coherence between comments and code through text-similarity techniques, and Khamis et al. (2010)

determine whether comments, specifically `@return` and `@param` comments, conform to particular format. Rabbi and Siddik (2020) propose a siamese network for correlating comment and code representations for this task.

There have also been some efforts for performing inconsistency detection upon code changes. Liu et al. (2018b) detect inconsistencies in a block/line comment upon changes to the corresponding code snippet using a random forest classifier with hand-engineered features. Stulova et al. (2020) concurrently present a preliminary study of an approach which maps a comment to the AST nodes of the method signature (before the code change) using BOW-based similarity metrics. This mapping is used to determine whether the code changes have triggered a comment inconsistency. Malik et al. (2008) predict whether a comment will be updated using a random forest classifier utilizing surface features that capture aspects of the method that is changed, the change itself, and ownership. They do not consider the existing comment since their focus is not inconsistency detection; instead, they aim to understand the rationale behind comment updating practices by analyzing useful features. Sadu (2019) develops at approach which locates inconsistent identifiers upon code changes through lexical matching rules. Svensson (2015) builds a system to mitigate the damage of inconsistent comments by prompting developers to validate a comment upon code changes. Comments that are not validated are identified, indicating that they may be out of date and unreliable. Nie et al. (2019) present a framework for maintaining consistency between code and todo comments by performing actions described in such comments when code changes trigger the specified conditions to be satisfied.

2.5. Bug Report Discussions

Software bugs that are observed in open-source projects are reported through bug and issue tracking systems like Bugzilla, Jira, and GitHub Issues. When a bug report is opened, developers engage in a discussion by posting comments to collectively understand the problem, diagnose the cause, and ultimately devise a solution (Arya et al., 2019; Noyori et al., 2019). The discussion can often be very long (Liu et al., 2020), encompassing comments from a number of different participants (Kavaler et al., 2017), and this deliberation can go on for extended periods of time (Kikas et al., 2015). Following the discussion, the bug is generally resolved by implementing the solution through code changes in the project's code base (Zhang et al., 2012). These changes can be implemented by core project members and other active contributors (Ye and Kishida, 2003), or less active developers, including peripheral developers (Krishnamurthy et al., 2016) and first-time contributors (Tan et al., 2020).

There have been a number of tasks that were proposed in order to streamline this process and consequently expedite bug resolution. This includes predicting severity (Lamkanfi et al., 2010; Chaturvedi and Singh, 2012; Tian et al., 2012a; Yang et al., 2014; Gomes et al., 2019; Arokiam and Bradbury, 2020), determining validity (Fan et al., 2020; He et al., 2020), detecting duplication (Tian et al., 2012b; Lazar et al., 2014; Aggarwal et al., 2015; Hindle and Onuczko, 2019), assigning relevant developers (Anvik, 2006; Baysal et al., 2009; Xi et al., 2018; Baloch et al., 2021), categorizing reports (Huang et al., 2011; Thung et al., 2012), and localizing the relevant "buggy" code within the code base (Saha et al., 2013; Rahman and Roy, 2018; Loyola et al., 2018; Zhu et al., 2020). There have also been efforts to better understand the contents of bug report discussions through sentiment analysis (Ding et al., 2018; Destefanis et al., 2018), language complexity analysis (Kavaler et al., 2017), summarization (Rastkar et al., 2014; Jiang et al., 2017; Li et al., 2018b; Liu et al., 2020), and dialogue act classification (Enayet and Sukthankar, 2020).

2.6. Dialogue + Software

There have been very limited work in building interactive AI tools for software engineering, with the exception of a few for a handful of tasks. This includes code generation (Chaurasia and Mooney, 2017; Gur et al., 2018; Yao et al., 2019) and query refinement for code search (Zhang et al., 2020). Wood et al. (2018) recently built a software-related dialogue corpus through a "Wizard of Oz" experiment to study the potential of a Q&A assistant during bug fixing. Lowe et al. (2015) developed a dialogue corpus based on Ubuntu chat logs to study Q&A assistants for technical support. Bradley et al. (2018) designed a voice-controlled conversational developer assistant which automates a sequence of low-level actions (e.g., Git commands) based on high-level intent, provided by the user.

2.7. Code Representations

To perform well on code-related tasks, neural models must learn to understand and generate source code representations. Some have represented code as a simple sequence of tokens (Iyer et al., 2016; Tufano et al., 2019b; Ahmad et al., 2020) while others have considered capturing structural properties of code (i.e., abstract syntax tree (AST), data flow, control flow) through tree-based (Rabinovich et al., 2017; Yin and Neubig, 2017; Alon et al., 2019, 2020; Sun et al., 2020; Chen et al., 2019; Wang et al., 2020b; Bui et al., 2021) and graph-based (Nguyen and Nguyen, 2015; Li et al., 2016; Allamanis et al., 2018b; Hellendoorn et al., 2020; Tarlow et al., 2020; Wang et al., 2020c; Mehrotra et al., 2021; Wei et al., 2020; LeClair et al., 2020; Abdelaziz et al., 2020; Yasunaga and Liang, 2020; Nair et al., 2020; Cummins et al., 2021) neural approaches.

With large pretrained language models leading to remarkable progress for numerous downstream tasks in NLP, it is no surprise that there are growing efforts to build analogous models for code. Following the ELMo framework (Peters et al., 2018), Karampatsis and Sutton (2020b) developed SCELMo. C-BERT (Buratti et al., 2020), CuBERT (Kanade et al., 2020), CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2021), and TreeBERT (Jiang et al., 2021b) all apply BERT-like (Devlin et al., 2019) training objectives to large amounts of code (and documentation in some cases) extracted from GitHub. PyMT5 (Clement et al., 2020) is pretrained much like T5 (Raffel et al., 2020). Ahmad et al. (2021) proposed PLBART, which was pretrained on a large amount of code from GitHub and software-related text from StackOverflow using BART-like (Lewis et al., 2020) training objectives. Inspired by GPT-2 (Radford et al., 2019), Svyatkovskiy et al. (2020) built GPT-C, and Lu et al. (2021) built CodeGPT. More recently, Chen et al. (2021) fine-tuned GPT-3 (Brown et al., 2020) on data from millions of GitHub repos to build Codex, which powers GitHub Copilot[1].

[1] `https://copilot.github.com/`

2.8. Handling Noise in Online Code Repositories

Though online code bases like GitHub and StackOverflow offer large volumes of data for code-related tasks, this data is often noisy (Allamanis, 2019; Yin et al., 2018). For instance, automatically collected data for the task of commit message generation can consist of poorly written commit messages (Etemadi and Monperrus, 2020). While deep learning models are robust to some level of noise, the *garbage in, garbage out* principle still holds (Geiger et al., 2020), in which having a large number of noisy examples impairs a model's ability to learn. So, training a model on too many examples with poor target commit messages can result in the model learning to generate low-quality commit messages. For more effective supervision and also for more accurate evaluation, automatically mined data from online code bases often need to be filtered to reduce noise. Iyer et al. (2016), Yin et al. (2018), and Yao et al. (2018) trained classifiers for this purpose on a manually annotated subset of data. Others filtered data using various task-specific heuristics (Allamanis et al., 2016; Hu et al., 2018; Fernandes et al., 2019; Allamanis, 2019). For instance, Allamanis et al. (2016) discard overridden methods for the task of method naming due to them having repetitive names.

2.9. Subtokenization

Source code often contains user-defined tokens, which causes the open vocabulary problem in this domain (Cvitkovic et al., 2019). Developers often write a code token as a combination of multiple subtokens which are conjoined through various techniques such as camel case (e.g., camelCase) and snake case (e.g., snake_case). By splitting a token into its subtokens (e.g., camelCase → camel, case; snake_case → snake, case), we are able to handle previously unseen tokens by exploiting patterns associated with individual subtokens, which are more likely to be seen. Moreover, even for tokens that are previously known, there may be substantial benefit in capitalizing on their composability in order to aggregate knowledge about their individual components and obtain a more comprehensive understanding. Subtokenization is used extensively in this domain for a number of different tasks (Allamanis et al., 2016; Alon et al., 2019; Fernandes et al., 2019).

Chapter 3

Associating Natural Language Comment and Source Code Entities

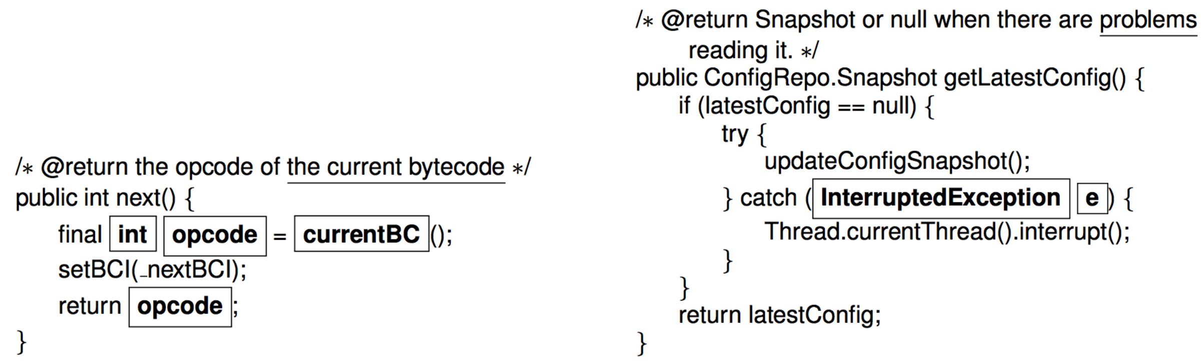
To keep comments in sync with the corresponding body of code, inconsistent comments which materialize as a result of code changes should be quickly detected and updated. Inconsistencies often emerge as a result of discrepancy between certain comment entities and certain code entities that have changed. In order to determine whether a particular comment entity becomes inconsistent upon changes to certain code entities and also how it should be updated to reflect these changes, we formulate a novel task which aims to learn explicit associations between entities in a comment and entities in the corresponding code. To perform this task, we design a set of highly salient features, which we later show to be useful for comment inconsistency detection (Chapter 4) and update (Chapter 5). Full details of this work are available in Panthaplackel et al. (2020a).

3.1. Task

Given a noun phrase (NP) in a comment, the task is to classify the relationship between the NP and each candidate code token in the corresponding source code as either associated or not associated. The candidate set includes all tokens other than select Java keywords (e.g., `try`, `public`, `throw`), operators (e.g., =), and symbols (e.g., brackets, parentheses). These elements are related to the programming language syntax and are commonly not described in comments. For instance, in Figure 3.1a, the tokens `int`, `opcode`, and `currentBC` are associated with the NP "the current bytecode" but `int` (the return type), `setBCI`, and `_nextBCI` are not.

This task shares similarities with anaphora resolution in natural language texts, including ones that explicitly refer to antecedents (coreference) as well as ones linked by associative relations (bridging anaphora) (Mitkov, 1999). In such a setting, the selected noun phrase within the comment is the anaphor, and tokens belonging to the source code serve as candidate antecedents. However our task is distinct from either in that it requires reasoning with respect to two different modalities (Allamanis et al., 2015; Loyola et al., 2017; Allamanis et al., 2018a). In Figure 3.1b, "problems" explicitly refers to `e`, but we need to know that `InterruptedException` is its type, which is a kind of `Exception`, and that `Exception` is a programming term for "problems."

Further, in our setting, an NP in the comment could be associated with multiple, distinct elements in the source code that do not belong to the same "chain." For these reasons, we frame our task broadly as *associating* a noun phrase in a natural language



```
/* @return the opcode of the current bytecode */
public int next() {
    final int opcode = currentBC();
    setBCI(_nextBCI);
    return opcode;
}
```

```
/* @return Snapshot or null when there are problems
     reading it. */
public ConfigRepo.Snapshot getLatestConfig() {
    if (latestConfig == null) {
        try {
            updateConfigSnapshot();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
    return latestConfig;
}
```

(a) Example from adriaanm-maxine-mirror.

(b) Example from node-sharing-plugin.

Figure 3.1: Examples of comment-code associations, with the boxed/bolded tokens in the code being associated with the underlined NP in the comment.

comment with individual code tokens in the corresponding body of code.

3.2. Data

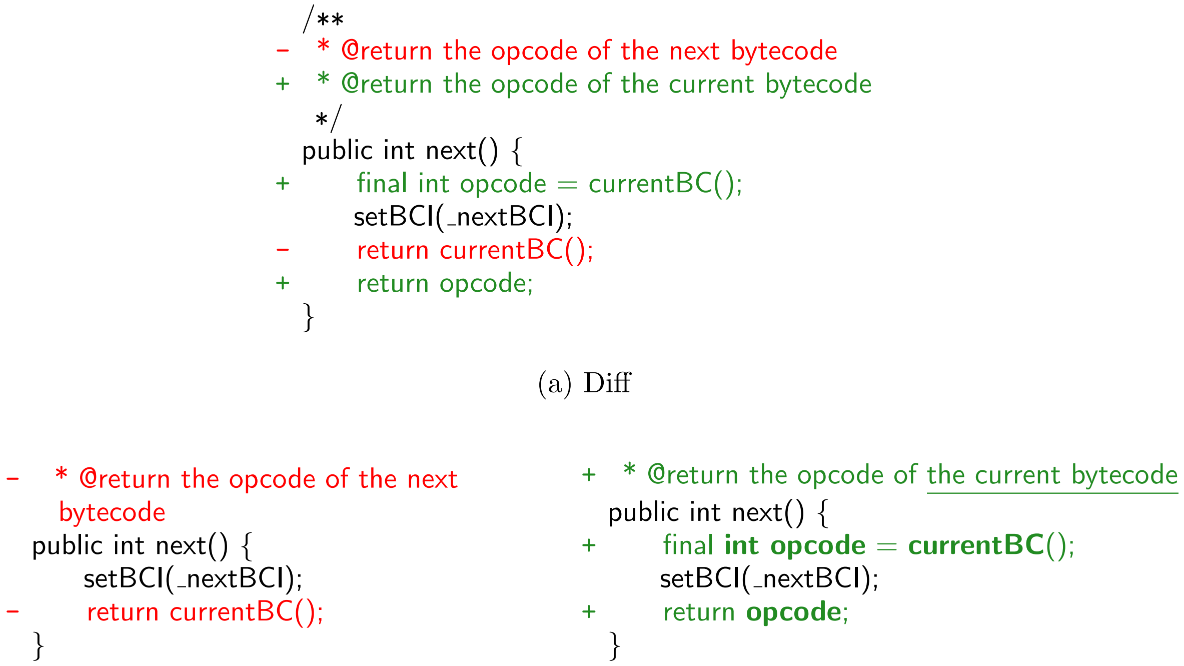
As an initial step towards learning these associations, we focus on Javadoc `@return`[1] comments, which serve to describe the return type and potential return values that are dependent on various conditions within a given method. Since these comments describe the output, which is computed by the various statements that make up the method, we find them to provide a fairly comprehensive overview of functionality. We also observe that `@return` comments tend to be more structured than other forms of comments, making it a cleaner data source and consequently, a reasonable starting point for the proposed task. We construct a dataset by extracting examples from all commits of popular open-source projects on GitHub. We rank the projects by the number of stars, and used the top ~1,000 projects, as they are considered to be of higher quality (Jarczyk et al., 2014). Each example we extract consists of a code change to a method body as well as a change to the corresponding `@return` comment.

3.2.1. Noisy Supervision

The core idea of our noisy supervision extraction method is to utilize revision histories from software version control systems (e.g., Git), based on prior research showing that source code and comments co-evolve (Fluri et al., 2007). Entities in comments have a higher chance of being associated with entities in source code if they were edited "at the same time", which can be approximated by "at the same commit". Therefore, mining such co-edits allow us to obtain noisy supervision for this task: we use the version control system Git to isolate parts of the code and comment that are added and deleted together.

We assign noisy labels to code tokens based on the intuition that parts of the code that are added are likely associated with the parts of the comment which are also added. Namely, we label code tokens in added lines in a given commit as associated with the NP that is introduced in the comment within the same commit, and we label all other code

[1] `https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html`



```
  /**
-  * @return the opcode of the next bytecode
+  * @return the opcode of the current bytecode
   */
  public int next() {
+      final int opcode = currentBC();
      setBCI(_nextBCI);
-      return currentBC();
+      return opcode;
  }
```

(a) Diff

```
-   * @return the opcode of the next
    bytecode
  public int next() {
      setBCI(_nextBCI);
-      return currentBC();
  }
```

(b) Before commit

```
+  * @return the opcode of the current bytecode
  public int next() {
+      final int opcode = currentBC();
      setBCI(_nextBCI);
+      return opcode;
  }
```

(c) After commit

Figure 3.2: Diff from a commit of the adriaanm-maxine-mirror project. Green lines starting with '+': added content; red lines starting with '-': removed. Based on the supervision provided by the diff, in Figure 3.2c, the bolded code tokens are automatically labeled as associated with the underlined NP in the comment.

tokens as not associated with the NP. These positive labels are noisy since a developer may also make other code changes that are not necessarily relevant to the NP that is added. On the other hand, the negative labels (not associated) have minimal noise, since code tokens in lines that are retained from the previous version of the code are unlikely to be associated with an NP that does not exist in the previous version of the comment.

3.2.2. Preprocessing and Filtering

We examine the two versions of the code and comment in a commit: *before* commit and *after* commit. We extract NPs from the two versions of the comments and compute their *diff*. We also compute the diff between the two versions of the code. We show these diffs in Figure 3.2, with added lines marked with "+" and deleted lines marked with "-".

From the diffs, we identify NPs which are unique to the *after* version of the comment and code tokens in the added lines of the *after* version of the code, and we obtain a pair in the form *(NPs, associated code tokens)*. We tokenize the full code sequence in the method corresponding to the *after* version and label any token that is not present in the associated code tokens as not associated. Following this procedure, each example consists of an NP and a sequence of labelled code tokens.

		Candidate Code Tokens		
	Examples	**Total**	**Unique**	**Average**
Train	776	23,188	5,908	29.9
Valid	77	2,488	911	32.3
Test	117	3,592	1,266	30.7

Table 3.1: Dataset statistics

After manually inspecting 200 examples, we impose constraints to filter out trivial cases, duplicates, and noisy examples, much like prior work (Section 2.8). Upon filtering, we partition our dataset into training, validation and test, as shown in Table 3.1.

The 117 examples in the test set were annotated by the first author. During pilot studies, two annotators jointly examined a sample set of method/comment pairs before converging on the criteria that were used for annotation. The standards used to identify a code token as associated include: whether it is directly referred to by the NP; it is an attribute, type, or method corresponding to the entity referred to by the NP; it is set equal to the entity referred to by the NP; and if an update to the NP would be required if the token is changed. To assess the quality of the annotations, we asked a graduate student, who is not one of the authors and has 5 years of Java experience, to annotate 286 code tokens (from 25 examples in the test set) that are labeled associated under the noisy supervision. The Cohen's kappa score between the two sets of annotations is 0.713, indicating satisfactory agreement.

3.3. Representations and Features

We design a set of features that encompasses surface features, word representations, code token representations, cosine similarity between terms, code structure, and the Java API. Our models leverage the 1,852-dimensional feature vector that results from concatenating these features.

Surface features: We incorporate two binary features, subtoken matching and presence in return statement, which we also use in two of the baseline models that are discussed in the next section. The subtoken matching feature indicates that a candidate code token matches exactly with a component of the given noun phrase, at the token-level or subtoken-level (ignoring case). The presence in return line feature indicates whether a candidate code token appears in a return statement or matches exactly with any token that appears in a return statement.

Word and code token representations: In order to derive representations of terms in the comment and code, we pre-train character-level and word-level embeddings for the comment and character-level, subtoken-level, and token-level embeddings for the code. These 128-dimensional embeddings are trained on a much larger corpus, consisting of 128,168 `@return` tag/Java method pairs that are extracted from GitHub. The pre-training task is to generate `@return` comments for Java methods using a single-layer, unidirectional sequence-to-sequence model (Sutskever et al., 2014). We use averaged embeddings to derive representations for the NP and candidate code token. Additionally, in order to provide a meaningful context, we average the embeddings corresponding to the full `@return` comment as well as the embeddings corresponding to the tokens in the same line in which the candidate token appears.

Cosine similarity: Recent work has used joint vector spaces for code/natural language description pairs and has shown that a body of code and its corresponding description have similar vectors (Gu et al., 2018). Since the content of `@return` comments often mention entities in the code, rather than modeling a joint vector space, we project the NP into the same vector space of the code by computing its vector representations with respect to the embeddings trained on Java code. We then compute the cosine similarity between the NP and the candidate code token at the token-level, subtoken-level, and character-level. The same procedure is followed to compute the cosine similarity between the NP and the line in the code on which the candidate code token appears.

Code structure: An abstract syntax tree (AST) captures the syntactic structure of a given body of code in tree form, as defined by Java's grammar. In order to represent properties of the candidate code token with respect to the overall structure of the method, we extract the node types of its parent and grandparent in the AST and represent them with one-hot encodings. This provides deeper insight into the role of a candidate code token within the broader context of the method by conveying details such as whether it appears within a method invocation, a variable declaration, a loop, an argument, a try/catch block, and so on.

Java API: We use one-hot encodings to represent features related to common Java types and the `java.util` package, which is a collection of utility classes, such as `List`, that we found to be used frequently. We hypothesize that these features could shed light into patterns that are exhibited by these frequently occurring tokens. To capture local context, we also include Java-related characteristics of code tokens adjacent to the candidate token such as whether it is a common Java type or one of the Java keywords.

3.4. Models

We develop two models representing different ways to tackle our proposed task: binary classification and sequence labeling. We also formulate multiple rule-based baselines.

3.4.1. Binary Classification

Given a sequence of code tokens and an NP in the comment, we independently classify each token as associated or not associated. Our classifier is a feedforward neural network with 4 fully-connected layers and a final output layer. As input, the network accepts a feature vector corresponding to the candidate code token (discussed in the previous section) and the model outputs a binary prediction for that token.

3.4.2. Sequence Labeling

Given a sequence of code tokens and an NP in the comment, we jointly classify the tokens regarding whether or not they are associated with the NP. The intuition behind structuring the problem this way is that the classification of a given code token can often depend on classifications of nearby tokens. For instance, in Figure 3.2, the `int` token that denotes the return type of the `next()` function is not associated with the specified NP, whereas the `int` token that is adjacent to `opcode` is considered to be associated because `opcode` is associated, and `int` is its type.

In order to re-establish the consecutive ordering of the original sequence, we inject removed Java keywords and symbols back into the sequence and introduce a third class which serves as the gold label for these inserted tokens. Specifically, we predict the three labels: *associated*, *not associated*, and a pseudo-label *Java*. Note that we disregard the classifications of these tokens during evaluation, i.e., if this pseudo-label is predicted for any other code token at test time, we automatically assign it to be not associated (on average, this happens ∼1% of the time). We construct a CRF model (Lample et al., 2016) by applying a neural CRF layer on top of a feedforward neural network that resembles that of the binary classifier in structure, except that the network accepts a matrix consisting of the feature vectors of all the tokens in the method.

3.4.3. Baselines

Random. Random classification of a code token as associated or not based on a uniform distribution.
Weighted random. Random classification of a code token as associated or not associated based on the probabilities of the associated and not associated classes as observed from the training set which are 42.8% and 57.2% respectively.
Subtoken matching. Any token for which the *subtoken matching* surface feature (introduced in the previous section) is set to be true is classified as associated while all other tokens are classified as not associated. Note that there will never be a case in which *all* associated code tokens will match at the token-level or subtoken-level with the noun phrase. We removed such trivial examples from the dataset during filtering because they can be resolved with simple string-matching tools and are not the focus of this work.
Presence in return statement. Any token for which the *presence in a return statement* surface feature (discussed in the previous section) is set to be true is classified as associated and all other tokens are classified as not associated.

3.5. Results and Discussion

The results of the four baselines and two learned models are given in Table 3.2. We compute precision, recall and F1 scores. Our analysis is primarily based on the results on the annotated test set; however, for completion, we present results on the unannotated test set in the full paper (Panthaplackel et al., 2020a).

Model	P	R	F1
Random	32.1	47.2	38.2
Weighted random	33.8	42.8	37.8
Subtoken matching	56.7	33.8	42.8
Presence in return line	51.5	45.8	48.5
Binary classifier	**57.4**	65.4	**61.0**
CRF	48.4	**66.3**	55.9

Table 3.2: Micro precision, recall, and F1 scores, evaluated on the annotated test set. The differences between F1 scores are statistically significant based on a signed rank t-test, with $p < 0.01$.

We observe that the subtoken matching and presence in return line heuristics provide some signal for this task, as the two corresponding baselines can do better than random. By incorporating these heuristics as features and combining them with other features, the binary classifier and CRF models can outperform all baselines by wide margins. This demonstrates the utility of our feature set in learning bimodal associations between comments and code.

Although the recall score of the CRF is slightly higher than that of the binary classifier, it is clear that the binary classifier performs better overall with respect to the F1 score. This may be due to the fact that the CRF requires additional parameters to model dependencies which may not be set accurately, given the limited amount of example-level data in our experimental setup.

Furthermore, while we expect the CRF to be more context-sensitive than the binary classifier, we do incorporate many contextual features (embeddings of surrounding and neighboring tokens, similarity of context with the NP, and Java API knowledge of neighboring tokens) with the binary classifier. With error analysis we found that the CRF model tends to make mistakes over tokens following Java keywords, as well as tokens that appear later in a method. This indicates that the CRF model could be struggling to reason over longer range dependencies and over longer sequences. Additionally, in contrast to the binary classification setting, Java keywords are present in the sequence labeling setting, so the CRF model must reason about many more code tokens than the binary classifier.

Chapter 4

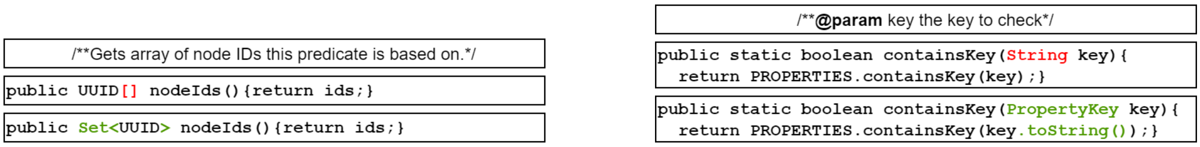
Just-In-Time Inconsistency Detection Between Comments and Source Code

To minimize the adverse effects of having comments which are out-of-sync with the corresponding body of code, there has been extensive work in automatically detecting inconsistent comments (Section 2.4). Prior work has predominantly focused on detecting inconsistencies that already reside within the code repository for a given software project. We refer to this as *post hoc inconsistency detection* since it occurs potentially many commits *after* the inconsistency has been introduced. Ideally, these inconsistencies should be detected before they ever enter the repository (e.g., during code review) since they pose a threat to the development cycle and reliability of the software until they are found. Because inconsistent comments generally arise as a consequence of developers failing to update comments immediately following code changes (Wen et al., 2019), we aim to detect whether a comment becomes inconsistent as a result of changes to the accompanying code, *before* these changes are merged into a code repository. We refer to this as *just-in-time inconsistency detection*, as it allows alerting developers of potential inconsistencies right before they can materialize. In Panthaplackel et al. (2021b), we develop a deep learning approach for just-in-time inconsistency detection that correlates a comment with changes in the corresponding body of code, which outperforms the post hoc setting.

4.1. Task

Suppose M_{old} from the consistent comment/method pair (C_{old}, M_{old}) is modified to M_{new}. If C_{old} is not in sync with M_{new} and is not updated, it will become inconsistent once M_{new} is committed. We frame this problem in two distinct settings, with the task being constant across both: determine whether C_{old} is inconsistent with M_{new}.

- **Post hoc:** Here, only the existing version of the comment/method pair is available; the code changes that triggered the inconsistency are unknown.
- **Just-in-time:** Here, the goal is to catch inconsistencies before they are committed. Detecting inconsistencies immediately following code changes allows us to utilize information from M_{old}. By considering how the changes affect the relationship the comment holds with the code, we can determine whether the comment remains consistent after the changes. For instance, in Figure 4.1a, the comment describes the return type of the `nodeIds()` as an array. When the method is modified to



```
/**Gets array of node IDs this predicate is based on.*/
public UUID[] nodeIds(){return ids;}
public Set<UUID> nodeIds(){return ids;}
```

(a) Inconsistent

```
/**@param key the key to check*/
public static boolean containsKey(String key){
  return PROPERTIES.containsKey(key);}
public static boolean containsKey(PropertyKey key){
  return PROPERTIES.containsKey(key.toString());}
```

(b) Consistent

Figure 4.1: In the example from the Apache Ignite project shown in Figure 4.1a, the existing comment becomes inconsistent upon changes to the corresponding method, and in the example from the Alluxio project shown in Figure 4.1b, the existing comment remains consistent after code changes.

return a `Set` instead of an array, the comment no longer describes the correct return type, making it inconsistent. Such analysis is not possible in post hoc inconsistency detection since the exact code changes that triggered inconsistency cannot be easily pinpointed, making it difficult to align the comment with relevant parts of the code.

4.2. Architecture

Prior work in post hoc inconsistency detection and the very few existing approaches in just-in-time inconsistency detection which exploit code changes rely on task-specific rules (Sadu, 2019), hand-engineered surface features (Liu et al., 2018b; Malik et al., 2008), and bag-of-words techniques (Liu et al., 2018b). Instead, we learn salient characteristics of the various inputs through a deep-learning framework that encodes their syntactic structures.

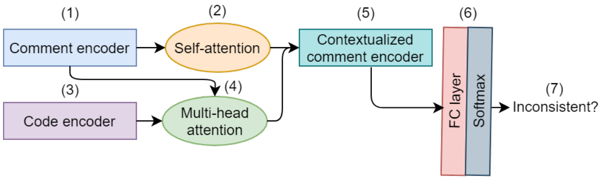


Figure 4.2: High-level architecture of our approach.

We aim to determine whether C_{old} is inconsistent by understanding its semantics and how it relates to M_{new} (or changes between M_{old} and M_{new}). We present an overview of our approach in Figure 4.2. First, the comment encoder, a BiGRU (Cho et al., 2014), encodes the sequence of tokens in C_{old} (Figure 4.2 (1)). When learning a representation for a given token, the forward and backward BiGRU passes provide context of other tokens in C_{old}, in principle. However, this information can get diluted, especially when there are long-range dependencies, and the relevant context can also vary across tokens. So, we update these representations from the comment encoder with more context about how they relate to the other tokens through multi-head self-attention (Vaswani et al., 2017) with hidden states of the comment encoder (Figure 4.2 (2)). Next, we learn code representations with a code encoder, which can be a sequence encoder or an abstract syntax tree (AST) encoder (Figure 4.2 (3)).

Since the essence of the task comes down to whether C_{old} accurately reflects M_{new}, we must capture the relationship between C_{old} and M_{new} (or changes between M_{old} and M_{new}). Prior work does this by computing comment/code similarity through lexical overlap rules (Ratol and Robillard, 2017; Sadu, 2019), which do not work well when different terms have similar meanings, and cosine similarity between vector representations, which have been found to perform poorly on their own (Liu et al., 2018b; Cimasa et al., 2019). Furthermore, this notion of similarity is only appropriate for the summary comment which provides an overview of the corresponding method as a whole. More specialized comment types like `@return` and `@param` describe only specific parts of the method, and thus their representations may not be very similar to the representation of the full method. We instead capture this relationship by computing multi-head attention between each hidden state of the comment encoder and the hidden states of the code encoder (Figure 4.2 (4)).

We combine the context vectors resulting from both attention modules to form enhanced representations of the tokens in C_{old}, which carry context from other parts of C_{old} as well as the code. These are then passed through another BiGRU encoder (Figure 4.2 (5)). We take the final state of this encoder to be the vector representation of the full comment, and we feed it through fully-connected and softmax layers (Figure 4.2 (6)). This leads to the final prediction (Figure 4.2 (7)).

4.2.1. Sequence Code Encoder

In the just-in-time setting, we represent the changes between M_{old} and M_{new} with M_{edit}, a *sequence of edit actions*, where each edit action is structured as `<Action> [span of tokens] <ActionEnd>`.[1] We define four types of edit actions: `Insert`, `Delete`, `Replace`, and `Keep`. Because the `Replace` action must simultaneously incorporate distinct content from two versions (i.e., tokens in the old version that will be replaced, and tokens in the new version that will take their place), it follows a slightly different structure:

```
<ReplaceOld> [span of old tokens]
<ReplaceNew> [span of new tokens]
<ReplaceEnd>
```

We encode M_{edit} with a BiGRU encoder. Because M_{old} is not available in the post hoc setting, we cannot construct an edit action sequence, and instead encode the sequence of tokens in M_{new} in this case.

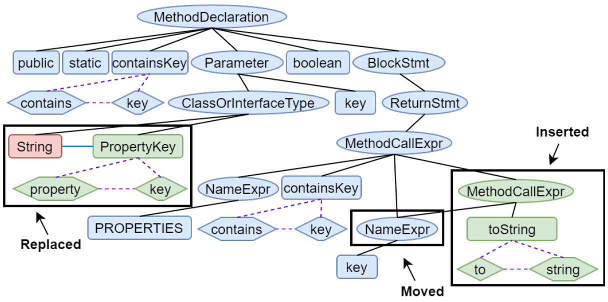


Figure 4.3: AST-based code edit representation (M_{edit}) corresponding to Figure 4.1b, with removed nodes in red and added nodes in green.

4.2.2. AST Code Encoder

To better exploit the syntactic structure of code, we leverage the abstract syntax tree (AST). Following prior work in other tasks (Fernandes et al., 2019; Yin et al., 2019), we encode ASTs and AST edits using gated graph neural networks (GGNNs) (Li et al., 2016). For the post hoc setting, we encode T, an AST-based representation corresponding to M_{new}. In the just-in-time setting, we instead encode T_{edit}, an AST-based edit representation. We compute AST node edits between T_{old} (corresponding to M_{old}) and T, identifying inserted, deleted, kept, replaced, and moved nodes. We merge the two, forming a unified representation, by consolidating identical nodes, as shown in Figure 4.3.

GGNN encoders for T and T_{edit} use *parent* (e.g., `public` → `MethodDeclaration`) and *child* (e.g., `MethodDeclaration` → `public`) edges. Like prior work (Fernandes et al., 2019), we add "subtoken nodes" for identifier leaf nodes to better handle previously unseen identifier names. To integrate these new nodes, we add *subnode* (e.g., `toString` → `to`), *supernode* (e.g., `to` → `toString`), *next subnode* (e.g., `to` → `string`), and *previous*

[1] Preliminary experiments showed that this performed better than structuring edits at the token-level as in other tasks (Shin et al., 2018; Li et al., 2018a; Dong et al., 2019; Awasthi et al., 2019).

subnode (e.g., `string` → `to`) edges. When encoding T_{edit}, we also include an *aligned* edge type between nodes in the two trees that correspond to an update (e.g., `String` and `PropertyKey`). Additionally, we learn *edit* embeddings for each action type. To identify how a node is edited (or not edited), we concatenate the corresponding edit embedding to its initial representation that is fed to the GGNN.

4.3. Data

In line with most prior work in inconsistency detection (Corazza et al., 2018; Tan et al., 2007, 2012; Khamis et al., 2010), we focus on identifying inconsistencies in comments comprising API documentation for Java methods. API documentation consists of two components: a main description and a set of tag comments (Oracle, 2020). While some have considered treating the full documentation as a single comment (Corazza et al., 2018), we choose to perform inconsistency detection at a more fine-grained level, analyzing individual comment types. Furthermore, in contrast to previous studies tailored to a specific type of tag (Zhou et al., 2017; Tan et al., 2012) or specific types of keywords and templates (Tan et al., 2007, 2011), we simultaneously consider multiple comment types with diverse characteristics. Namely, we address inconsistencies in the `@return` tag comment, which describes a method's return type, and the `@param` tag comment, which describes an argument of the method. Additionally, we examine inconsistencies in the less-structured summary comment, which comes from the first sentence of the main description.

By detecting inconsistencies at the time of code change, we can extract automatic supervision from commit histories of open-source Java projects. Namely, we compare consecutive commits, collecting instances in which a method is modified. We extract the comment/method pairs from each version: (C_1, M_1), (C_2, M_2). By assuming that the developer updated the comment because it would have otherwise become inconsistent as a result of code changes, we take C_1 to be inconsistent with M_2, consequently leading to a *positive example*, with C_{old}=C_1, M_{old}=M_1, and M_{new}=M_2. For *negative examples*, we additionally examine cases in which C_1=C_2 and assume that if the existing comment would have become inconsistent, the developer would have updated it. Following this process, we collect `@return`, `@param`, and summary comment examples.

To minimize noise, we filter the data by applying heuristics (Section 2.8). In line with prior work (Ren et al., 2019; Movshovitz-Attias and Cohen, 2013), we consider a cross-project setting with no overlap between the projects from which examples are extracted in training/-validation/test sets. From our data collection procedure, we obtain substantially more negative examples than positive ones, which is not surprising because many changes do not require comment updates (Wen et al., 2019). We downsample negative examples, for each partition and comment type, to construct a balanced dataset. Statistics of our final dataset are shown in Table 4.1. For more reliable evaluation, we curate a clean a sample of 300 examples (corresponding to 101 projects) from the test set, consisting of 50 positive and 50 negative examples of each comment type. Note that we subtokenize M_{new}, and M_{edit} (Section 2.9). Since comments often

	Train	**Valid**	**Test**	**Total**
@return	15,950	1,790	1,840	19,580
@param	8,640	932	1,038	10,610
Summary	8,398	1,034	1,066	10,498
Full	32,988	3,756	3,944	40,688
Projects	829	332	357	1,518

Table 4.1: Data partitions

include code tokens, we also subtokenize C_{old}.

4.4. Models

We outline baseline, post hoc, and just-in-time inconsistency detection models.

4.4.1. Baselines

Lexical overlap: A comment often has lexical overlap with the corresponding method. We include a rule-based just-in-time baseline, $\textsc{Overlap}(C_{old}, \text{deleted})$, which classifies C_{old} as inconsistent if at least one of its tokens matches a code token belonging to a `Delete` or `ReplaceOld` span in M_{edit}.

Corazza et al. (2018): This post hoc bag-of-words approach classifies whether a comment is coherent with the method that it accompanies using an SVM with TF-IDF vectors corresponding to the comment and method. We simplify the original data pre-processing, but validate that the performance matches the reported numbers.

CodeBERT BOW: We develop a more sophisticated bag-of-words baseline that leverages pretrained CodeBERT (Feng et al., 2020) embeddings. These embeddings were pretrained on a large corpus of natural language/code pairs. In the post hoc setting, we consider CodeBERT BOW (C_{old}, M_{new}), which computes the average embedding vectors of C_{old} and M_{new}. These vectors are concatenated and fed through a feedforward network. In the just-in-time setting, we compute the average embedding vector of M_{edit} rather than M_{new}, and we refer to this baseline as CodeBERT BOW (C_{old}, M_{edit}).

Liu et al. (2018b): This is a just-in-time approach for detecting whether a block/line comment becomes inconsistent upon changes to the corresponding code snippet. Their task is slightly different as block/line comments describe low-level implementation details and generally pertain to only a limited number of lines of code, relative to API comments. However, we consider it as a baseline since it is closely related. They propose a random forest classifier which leverages features which capture aspects of the code changes (e.g., whether there is a change to a `while` statement), the comment (e.g., number of tokens), and the relationship between the comment and code (e.g., cosine similarity between representations in a shared vector space). We re-implemented this approach based on specifications in the paper, as their code was not publicly available. We disregard 9 (of 64) features that are not applicable in our setting.

4.4.2. Our Models

Post hoc: We consider three models, with different ways of encoding the method. $\textsc{Seq}(C_{old}, M_{new})$ encodes M_{new} with a GRU, $\textsc{Graph}(C_{old}, T)$ encodes T with a GGNN, and $\textsc{Hybrid}(C_{old}, M_{new}, T)$ uses both. Multi-head attention in $\textsc{Hybrid}(C_{old}, M_{new}, T)$ is computed with the hidden states of the two encoders separately and then combined.[2]

[2] More complex hybrid approaches for combining sequence and graph representations did not help for our task (Fernandes et al., 2019; Hellendoorn et al., 2020).

Just-In-Time: To allow fair comparison with the post hoc setting, these models are identical in structure to the models described above except that M_{edit} is used instead of M_{new}.

Just-In-Time + features: Because injecting explicit knowledge can boost the performance of neural models (Chen et al., 2017; Xuan et al., 2018), we investigate adding linguistic and lexical features to our approach. In Section 3.3, we identified a set of features which were useful for learning to associate comments and code. By design, components of our architecture encompass some of these features. For instance, we derive token representations for code and comments through embeddings and learned encoder representations, the GGNN captures code structure, and attention addresses similarity between comment and code representations to some extent. We specifically incorporate surface features, some Java-related features, and a handful of additional features which appear relevant to the task based on our inspection of the data. These features, which are computed at the subtoken/subnode-level, are concatenated to M_{edit} and C_{old} embeddings and then passed through a linear layer, before providing them as inputs to the encoders.

- **Features specific to C_{old}:** Motivated by the *subtoken matching* feature from Section 3.3, we include whether a subtoken matches a code subtoken that is inserted, deleted, or replaced in M_{edit}. By aligning parts of C_{old} with code edits, these features assist the model in identifying subtokens in C_{old} which are important for the task. In order to exploit common patterns for different types of subtokens, we incorporate features that identify whether the subtoken appears more than once in C_{old} or is a stop word, and its part-of-speech.

- **Features specific to M_{edit}:** We apply the subtoken matching feature to subtokens in M_{edit} as well to indicate whether the subtoken matches a subtoken in C_{old}. This is intended to provide additional signal for highlighting specific locations in M_{edit} which may be directly relevant to C_{old}. Next, we aim to take advantage of common patterns among different types of code subtokens by incorporating features that identify certain categories: edit keywords, Java keywords, and operators. If a token is not an edit keyword, we have indicator features for whether it is part of a `Insert`, `Delete`, `ReplaceNew`, `ReplaceOld`, or `Keep` span. We believe this will be particularly helpful for longer spans since edit keywords only appear at either the beginning or end of a span.

- **Shared features:** We incorporate the *presence in return statement* feature from Section 3.3, ie., whether a given subtoken matches a subtoken in a return statement. Since there are two versions of the code, we include 3 separate features corresponding to presence in a return statement unique to M_{old}, unique to M_{new}, and present in both. Similarly, we indicate whether the subtoken matches a subtoken in the `return` type that is unique to M_{old}, unique to M_{new}, or present in both. Finally, we include whether a subtoken was originally split from a larger token and its index if so (e.g., split from `camelCase`, `camel` and `case` are subtokens with indices 0 and 1 respectively). These features aim to encode important relationships between adjacent tokens that are lost once the body of code and comment are transformed into a single, subtokenized sequences.

	Cleaned Test Sample				Full Test Set			
Model	**P**	**R**	**F1**	**Acc**	**P**	**R**	**F1**	**Acc**
Baselines								
Overlap(C_{old}, deleted)	77.7	72.0	74.7	75.7	74.1	62.8	68.0	70.4
Corazza et al. (2018)	65.1	46.0	53.9	60.7	63.7	47.8	54.6	60.3
CodeBERT BOW (C_{old}, M_{new})	66.2	70.4	67.9	66.9	68.9	73.2	70.7	69.8
CodeBERT BOW (C_{old}, M_{edit})	65.5	80.9	72.3	69.0	67.4	76.8	71.6	69.6
Liu et al. (2018b)	77.6	74.0	75.8	76.3	77.5	63.8	70.0	72.6
Post hoc								
Seq(C_{old}, M_{new})	58.9	68.0	63.0	60.3	60.6	73.4	66.3	62.8
Graph(C_{old}, T)	60.6	70.2	65.0	62.2	62.6	72.6	67.2	64.6
Hybrid(C_{old}, M_{new}, T)	53.7	77.3	63.3	55.2	56.3	80.8	66.3	58.9
Just-In-Time								
Seq(C_{old}, M_{edit})	83.8	79.3	81.5	82.0	80.7	73.8	77.1	78.0
Graph(C_{old}, T_{edit})	84.7	78.4	81.4	82.0	79.8	74.4	76.9	77.6
Hybrid(C_{old}, M_{edit}, T_{edit})	87.1	79.6	83.1	83.8	80.9	74.7	77.7	78.5
Just-In-Time + features								
Seq(C_{old}, M_{edit}) + features	91.3	82.0	86.4	87.1	88.4	73.2	80.0	**81.8**
Graph(C_{old}, T_{edit}) + features	85.8	**87.1**	86.4	86.3	83.8	**78.3**	**80.9**	81.5
Hybrid(C_{old}, M_{edit}, T_{edit}) + features	**92.3**	82.4	**87.1**	**87.8**	**88.6**	72.4	79.6	81.5

Table 4.2: Results for baselines, post hoc, and just-in-time models. Differences in F1 and Acc between just-in-time vs. baseline models, just-in-time vs. post hoc models, and just-in-time + features vs. just-in-time models are statistically significant.

4.5. Results and Discussion

We report common classification metrics: precision, recall, and F1 (w.r.t. the positive label) and accuracy (averaged across 3 random restarts). We also perform significance testing (Berg-Kirkpatrick et al., 2012).

In Table 4.2, we report results for baselines, post hoc and just-in-time inconsistency detection models. In the post hoc setting, we find that our three models can achieve higher F1 scores than the bag-of-words approach proposed by Corazza et al. (2018); however, they underperform the CodeBERT BOW (C_{old}, M_{new}) baseline and significantly underperform all just-in-time models, including the simple rule-based baseline. This demonstrates the benefit of performing inconsistency detection in the just-in-time setting, in which the code changes that trigger inconsistency are available. Additionally, by encoding the syntactic structures of the comment and code changes, our just-in-time models outperform this rule-based baseline as well as all other baselines and post hoc approaches. While the Hybrid(C_{old}, M_{edit}, T_{edit}) model achieves slightly higher scores (on the basis of F1 and accuracy) than Seq(C_{old}, M_{edit}) and Graph(C_{old}, T_{edit}), the differences are not statistically significant.

Our just-in-time models outperform the rule-based and feature-based baselines, without any hand-engineered rules or features. However, by incorporating surface features into our just-in-time models, we can further boost performance (by statistically significant margins). This suggests that our approach can be used in conjunction with task-specific rules (Tan et al., 2007, 2011, 2012; Ratol and Robillard, 2017) and feature sets (Liu et al., 2018b) to build improved systems for specific domains.

Furthermore, in Table 4.3, we analyze the performance of the three just-in-time + features models with respect to individual comment types. While these models are trained on all comment types together without explicitly tailoring it in any way to handle them

		Cleaned Test Sample			
	Model	**P**	**R**	**F1**	**Acc**
`@return`	Seq(C_{old}, M_{edit}) + features	88.5*	72.0*	**79.4***	**81.3***
	Graph(C_{old}, T_{edit}) + features	81.2	**77.3**	79.1*	79.7
	Hybrid(C_{old}, M_{edit}, T_{edit}) + features	**88.7***	72.0*	**79.4***	**81.3***
`@param`	Seq(C_{old}, M_{edit}) + features	90.0	**95.3**	92.5	92.3†
	Graph(C_{old}, T_{edit}) + features	**96.5**	92.0	**94.2**	**94.3**
	Hybrid(C_{old}, M_{edit}, T_{edit}) + features	94.6	89.3	91.8	92.0†
Summary	Seq(C_{old}, M_{edit}) + features	**96.0**	78.7	86.5§	87.7
	Graph(C_{old}, T_{edit}) + features	80.8	**92.0**	86.0§	85.0
	Hybrid(C_{old}, M_{edit}, T_{edit}) + features	93.7	86.0	**89.5**	**90.0**

Table 4.3: Evaluating performance with respect to different types of comments. Scores are averaged across 3 random restarts, and scores for which the difference in performance is *not* statistically significant are shown with identical symbols.

differently, they are still able to achieve reasonable performance across types. We provide further analysis of individual comment types and compare to comment-specific baselines in the full paper (Panthaplackel et al., 2021b).

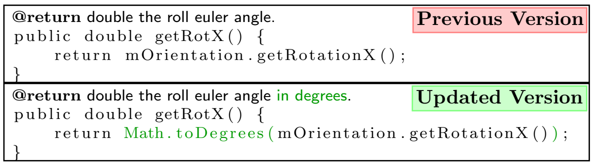
Chapter 5

Updating Natural Language Comments Based on Code Changes

Once inconsistent comments are detected upon code changes, the next step is to *update* them to reflect these changes. To guide developers with this, we aim to generate suggestions for updated comments. In principle, we could do this by generating a completely new comment that corresponds to the most recent version of the code through the extensive work in comment generation (Section 2.1). However, this discards potentially salient content from the existing comment and also fails to consider the code changes which could point to critical aspects of the code that should be highlighted in the updated comment. Therefore, we formulate the novel task of learning to update an existing comment based on changes to the corresponding body of code. This task is intended to align with how developers edit a comment when they introduce changes in the corresponding code. Rather than deleting it and starting from scratch, they would likely only modify the specific parts relevant to the code changes. We replicate this process through a novel approach which is designed to correlate edits across two distinct language representations: source code and natural language comments. Full details of this work are available in Panthaplackel et al. (2020b).

5.1. Task

Given a method, its corresponding comment, and an updated version of the method, the task is to update the comment so that it is consistent with the code in the new method. For the example in Figure 5.1, we want to generate *"`@return` double the roll euler angle in degrees."* based on the changes between the two versions of the method and the existing comment *"`@return` double the roll euler angle."* Concretely, given (M_{old}, C_{old}) and M_{new}, where M_{old} and M_{new} denote the old and new versions of the method, and C_{old} signifies the previous version of the comment, the task is to produce C_{new}, the updated version of the comment.



Previous Version

```
@return double the roll euler angle.
public double getRotX() {
    return mOrientation.getRotationX();
}
```

Updated Version

```
@return double the roll euler angle in degrees.
public double getRotX() {
    return Math.toDegrees(mOrientation.getRotationX());
}
```

Figure 5.1: Changes in the `getRotX` method and its corresponding `@return` comment between two subsequent commits of the rajawali project.

5.2. Edit Model

We design a system that examines source code changes and how they relate to the existing comment in order to produce an updated comment that reflects the code modifications. Figure 4.2 shows a high-level overview of our system.

5.2.1. Encoders

Using the edit lexicon defined in Section 4.2.1, we unify M_{old} and M_{new} into a single *diff* sequence that explicitly identifies code edits, M_{edit}. We encode this sequence with a BiGRU encoder (top right of Figure 5.2). We encode the existing comment (C_{old}) with another BiGRU encoder (top left). To better learn associations between comment and code entities, we also include the linguistic and lexical features discussed in Section 4.4.2. We incorporate these features into the network the same way as before.

5.2.2. Decoder

The decoder also takes the form of a GRU. Since C_{old} and C_{new} are closely related, training the decoder to directly generate C_{new} risks having it learn to just copy C_{old}. To explicitly inform the decoder of edits, we define the target output as a sequence of edit actions, C_{edit}, indicating how the existing comment should be revised.

For representing C_{edit}, we introduce a slightly modified set of specifications that disregards the `Keep` type when constructing the sequence of edit actions, referred to as a *condensed edit sequence*. The intuition for disregarding `Keep` and the span of tokens to which it applies is that we can simply copy the content that is retained between C_{old} and C_{new}, instead of generating it anew. By doing post hoc copying, we simplify learning for the model since it has to only learn *what to change* rather than also having to learn *what to keep*. We design a method to deterministically place edits in their correct positions in the absence of `Keep` spans. For the example in Figure 5.1, the raw sequence `<Insert>in degrees<InsertEnd>` does not encode information as to where "in degrees" should be inserted. To address this, we bind an insert sequence with the minimum number of words (aka "anchors") such that the place of insertion can be uniquely identified. This results in the structure that is shown for C_{edit} in Figure 4.2. Here "angle" serves as the anchor point, identifying the insert location. Following the structure of `Replace`, this sequence indicates that "angle" should be replaced with "angle in degrees," effectively inserting "in degrees" and keeping "angle" from C_{old}, which appears immediately before the insert location.

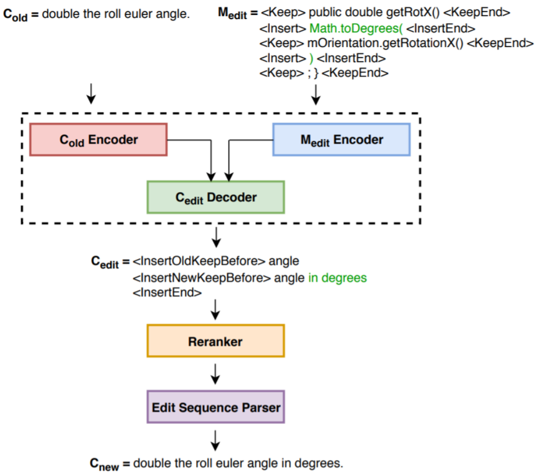

C_old = double the roll euler angle.
M_edit = <Keep> public double getRotX() <KeepEnd>
<Insert> Math.toDegrees(<InsertEnd>
<Keep> mOrientation.getRotationX() <KeepEnd>
<Insert>) <InsertEnd>
<Keep> ; } <KeepEnd>
C_old Encoder
M_edit Encoder
C_edit Decoder
C_edit = <InsertOldKeepBefore> angle
<InsertNewKeepBefore> angle in degrees
<InsertEnd>
Reranker
Edit Sequence Parser
C_new = double the roll euler angle in degrees.

Figure 5.2: High-level architecture of our approach.

The decoder essentially has three subtasks: (1) identify edit locations in C_{old}; (2) determine parts of M_{edit} that pertain to making these edits; and (3) apply updates in the given locations based on the relevant code changes. We rely on an attention mechanism (Luong et al., 2015) over the hidden states of the two encoders to accomplish the first two goals. At every decoding step, rather than aligning the current decoder state with all the encoder hidden states jointly, we align it with the hidden states of the two encoders separately. We concatenate the two resulting context vectors to form a unified context vector that is used in the final step of computing attention, ensuring that we incorporate pertinent content from both input sequences. Consequently, the resulting attention vector carries information relating to the current decoder state as well as knowledge aggregated from relevant portions of C_{old} and M_{edit}.

Using this information, the decoder performs the third subtask, which requires reasoning across language representations. Specifically, it must determine how the source code changes that are relevant to the current decoding step should manifest as natural language updates to the relevant portions of C_{old}. At each step, it decides whether it should begin a new edit action by generating an edit start keyword, continue the present action by generating a comment token, or terminate the present action by generating an end-edit keyword. Because actions relating to deletions will include tokens in C_{old}, and actions relating to insertions are likely to include tokens in M_{edit}, we equip the decoder with a pointer network (Vinyals et al., 2015) to accommodate copying tokens from C_{old} and M_{edit}. The decoder generates a sequence of edit actions, which will have to be parsed into a comment.

5.2.3. Parsing Edit Sequences

Since the decoder is trained to predict a sequence of edit actions, we must align it with C_{old} and copy unchanged tokens in order to produce the edited comment during inference. We denote the predicted edit sequence as C'_{edit} and the corresponding parsed output as C'_{new}. This procedure entails simultaneously following pointers, left-to-right, on C_{old} and C'_{edit}, which we refer to as P_{old} and P_{edit} respectively. P_{old} is advanced, copying the current token into C'_{new} at each point, until an edit location is reached. The edit action corresponding to the current position of P_{edit} is then applied, and the tokens from its relevant span are copied into C'_{new} if applicable. Finally, P_{edit} is advanced to the next action, and P_{old} is also advanced to the appropriate position in cases involving deletions and replacements. This process repeats until both pointers reach the end of their respective sequences.

5.2.4. Reranking

Reranking allows the incorporation of additional priors that are difficult to back-propagate, by re-scoring candidate sequences during beam search (Neubig et al., 2015; Ko et al., 2019; Kriz et al., 2019). We incorporate two heuristics to re-score the candidates: 1) generation likelihood and 2) similarity to C_{old}. These heuristics are computed after parsing the candidate edit sequences (Section 5.2.3).

Generation likelihood: Since the edit modelis trained on edit actions only, it does not globally score the resulting comment in terms of aspects such as fluency and overall suitability for the updated method. To this end, we make use of a pre-trained comment generation model (Section 5.4.2) that is trained on a substantial amount of data for generating C_{new} given only M_{new}. We compute the length-normalized probability of this model generating the parsed candidate comment, C'_{new}, (i.e., $P(C'_{new} \mid M_{new})^{1/N}$ where N is the number of tokens in C'_{new}). This model gives preference to comments that are more likely for M_{new} and are more consistent with the general style of comments.

Similarity to C_{old}: So far, our model is mainly trained to produce accurate edits;

however, we also follow intuitions that edits should be minimal (as an analogy, the use of Levenshtein distance in spelling correction). To give preference to predictions that accurately update the comment with minimal modifications, we use similarity to C_{old} as a heuristic for reranking. We measure similarity between the parsed candidate prediction and C_{old} using METEOR (Banerjee and Lavie, 2005).

Reranking score: The reranking score for each candidate is a linear combination of the original beam score, the generation likelihood, and the similarity to C_{old} with coefficients 0.5, 0.3, and 0.2 respectively (tuned on validation data).

5.3. Data

As a first step, we focus on performing this task on `@return` comments, which we find to follow a well-defined structure and describe characteristics of the output of a method (Section 3.2). We use the subset of examples corresponding to *positive* `@return` examples from the dataset we introduced in Section 4.3, in which the method and comment are simultaneously changed between two consecutive commits. We provide dataset statistics in Table 5.1.

		Train	**Valid**	**Test**
	Examples	5,791	712	736
	Projects	526	274	281
	Edit Actions	8,350	1,038	1,046
	Sim (M_{old}, M_{new})	0.773	0.778	0.759
	Sim (C_{old}, C_{new})	0.623	0.645	0.635
Code	Unique	7,271	2,473	2,690
	Mean	86.4	87.4	97.4
	Median	46	49	50
Comm.	Unique	4,823	1,695	1,737
	Mean	10.8	11.2	11.1
	Median	8	9	9

Table 5.1: Number of examples, projects, and edit actions; average similarity between M_{old} and M_{new} as the ratio of overlap to average sequence length; average similarity between C_{old} and C_{new} as the ratio of overlap to average sequence length; number of unique code tokens and mean and median number of tokens in a method; and number of unique comment tokens and mean and median number of tokens in a comment.

5.4. Experimental Method

We evaluate our approach against multiple rule-based baselines and comment generation models.

5.4.1. Baselines

Copy: Since much of the content of C_{old} is typically retained in the update, we include a baseline that merely copies C_{old} as the prediction for C_{new}.

Return type substitution: The return type of a method often appears in its `@return` comment. If the return type of M_{old} appears in C_{old} and the return type is updated in the code, we substitute the new return type while copying all other parts of C_{old}. Otherwise, C_{old} is copied as the prediction.

Return type substitution w/ null handling: As an addition to the previous method, we also check whether the token `null` is added to either a `return` statement or `if` statement in the code. If so, we copy C_{old} and append the string *or null if null*, otherwise, we simply copy C_{old}. This baseline addresses a pattern we observed in the data in which ways to handle `null` input or cases that could result in `null` output were added.

5.4.2. Generation Model

One of our main hypotheses is that modeling edit sequences is better suited for this task than generating comments from scratch. However, a counter argument could be that a comment generation model could be trained from substantially more data, since it is much easier to obtain parallel data in the form (method, comment), without the constraints of simultaneous code/comment edits. Hence the power of large-scale training could out-weigh edit modeling. To this end, we compare with a generation model trained on 103,473 method/`@return` comment pairs collected from GitHub.

We use the same underlying neural architecture as our edit model to make sure that the difference in results comes from the amount of training data and from using edit of representations only: a two-layer, BiGRU that encodes the sequence of tokens in the method, and an attention-based GRU decoder with a copy mechanism that decodes a sequence of comment tokens. Evaluation is based on the 736 (M_{new}, C_{new}) pairs in the test set described in Section 5.3. We ensure that the projects from which training examples are extracted are disjoint from those in the test set, adhering to our cross-project partitioning strategy (Section 4.3).

5.4.3. Reranked Generation Model

In order to allow the generation model to exploit the old comment, this system uses similarity to C_{old} (Section 5.2.4) as a heuristic for reranking the top candidates from the previous model. The reranking score is a linear combination of the original beam score and the METEOR score between the candidate prediction and C_{old}, both with coefficient 0.5 (tuned on validation data).

5.5. Evaluation

We evaluate models using automated metrics and human evaluation.

5.5.1. Automatic Evaluation

Metrics: We compute exact match, i.e., the percentage of examples for which the model prediction is identical to the reference comment C_{new}. This is often used to evaluate tasks involving source code edits (Shin et al., 2018; Yin et al., 2019). We also report two prevailing language generation metrics: METEOR (Banerjee and Lavie, 2005), and average sentence-level BLEU-4 (Papineni et al., 2002) that is previously used in code-language tasks (Iyer et al., 2016; Loyola et al., 2017).

Previous work suggests that BLEU-4 fails to accurately capture performance for tasks related to edits, such as text simplification (Xu et al., 2016), grammatical error correction (Napoles et al., 2015), and style transfer (Sudhakar et al., 2019), since a system that merely copies the input text often achieves a high score. Therefore, we also include two text-editing metrics to measure how well our system learns to *edit*: SARI (Xu et al., 2016), originally proposed to evaluate text simplification, is essentially the average of N-gram F1 scores corresponding to add, delete, and keep edit operations;[1] GLEU (Napoles et al., 2015), used in grammatical error correction and style transfer, takes into account

[1] Although the original formulation only used precision for the delete operation, more recent work computes F1 for this as well (Dong et al., 2019; Alva-Manchego et al., 2019).

	xMatch (%)	METEOR	BLEU-4	SARI	GLEU
Baselines					
Copy	0.000	34.611	46.218	19.282	35.400
Return type subt.	13.723§	43.106¶	50.796$^{\|}$	31.723	42.507*
Return type subst. + null	13.723§	43.359	**51.160**†	32.109	42.627*
Non-reranked models					
Generation	1.132	11.875	10.515	21.164	17.350
Edit	17.663	42.222¶	48.217	**46.376**	45.060
Reranked models					
Generation	2.083	18.170	18.891	25.641	22.685
Edit	**18.433**	**44.698**	50.717$^{\|\dagger}$	45.486	**46.118**

Table 5.2: Exact match, METEOR, BLEU-4, SARI, and GLEU scores. Scores for which the difference in performance is *not* statistically significant are shown with identical symbols.

the source sentence and deviates from BLEU by giving more importance to n-grams that have been correctly changed.

Results: We report automatic metrics averaged across three random initializations for all learned models, and use bootstrap tests (Berg-Kirkpatrick et al., 2012) for statistical significance (with $p < 0.05$). Table 5.2 presents the results. While reranking using C_{old} appears to help the generation model, it still substantially underperforms all other models, across all metrics. Although this model is trained on considerably more data, it does not have access to C_{old} during training and uses fewer inputs and consequently has less context than the edit model. Reranking slightly deteriorates the edit model's performance with respect to SARI; however, it provides statistically significant improvements on most other metrics.

Although two of the baselines achieve slightly higher BLEU-4 scores than our best model, these differences are not statistically significant, and our model is better at *editing* comments, as shown by the results on exact match, SARI, and GLEU. In particular, our edit models beat all other models with wide, statistically significant, margins on SARI, which explicitly measures performance on edit operations. Furthermore, merely copying C_{old}, yields a relatively high BLEU-4 score of 46.218. The *return type substitution* and *return type substitution w/ null handling* baselines produce predictions that are identical to C_{old} for 74.73% and 65.76% of the test examples, respectively, while it is only 9.33% for the reranked edit model. In other words, the baselines attain high scores on automatic metrics and even beat our model on BLEU-4, without actually performing edits on the majority of examples. This further underlines the shortcomings of some of these metrics and the importance of conducting human evaluation for this task.

5.5.2. Human Evaluation

Automatic metrics often fail to incorporate semantic meaning and sentence structure in evaluation as well as accurately capture performance when there is only one gold-standard reference; indeed, these metrics do not align with human judgment in other generation tasks like grammatical error correction (Napoles et al., 2015) and dialogue generation (Liu et al., 2016). Since automatic metrics have not yet been explored in the context of the new task we are proposing, we find it necessary to conduct human evaluation and study whether these metrics are consistent with human judgment.

Setup: Our study aims to reflect how a comment update system would be used in practice, such as in an Integrated Development Environment (IDE). When developers change code, they would be shown suggestions for updating the existing comment. If they think the comment needs to be updated to reflect the code changes, they could select the one that is most suitable for the new version of the code or edit the existing comment themselves if none of the options are appropriate.

We simulated this setting by asking a user to select the most appropriate updated comment from a list of suggestions, given C_{old} as well as the *diff* between M_{old} and M_{new} displayed using GitHub's diff interface. The user can select multiple options if they are equally good or a separate *None* option if no update is needed or all suggestions are poor.

The list of suggestions consists of up to three comments, predicted by the strongest benchmarks and our model : (1) return type substitution w/ null handling, (2) reranked generation model, and (3) reranked edit model, arranged in randomized order. We collapse identical predictions into a single suggestion and reward all associated models if the user selects that comment. Additionally, we remove any prediction that is identical to C_{old} to avoid confusion as the user should never select such a suggestion. We excluded 6 examples from the test set for which all three models predicted C_{old} for the updated comment.

Nine students (8 graduate/1 undergraduate) and one full-time developer at a large software company, all with 2+ years of Java experience, participated in our study. To measure inter-annotator agreement, we ensured that every example was evaluated by two users. We conducted a total of 500 evaluations, across 250 distinct test examples.

Results: Table 5.3 presents the percentage of annotations (out of 500) for which users selected comment suggestions that were produced by each model. Using Krippendorff's α (Krippendorff, 2011) with MASI distance (Passonneau, 2006) (which accommodates our multi-label setting), inter-annotator agreement is 0.64, indicating satisfactory agreement. The reranked edit model beats the strongest baseline and reranked generation by wide statistically-significant margins. From rationales provided by two annotators, we observe that some options were not selected because they removed relevant information from the existing comment, and not surprisingly, these options often corresponded to the comment generation model.

Baseline	**Generation**	**Edit**	**None**
18.4%	12.4%	30.2%	55.0%

Table 5.3: Percentage of annotations for which users selected comment suggestions produced by each model. All differences are statistically significant.

Users selected none of the suggested comments 55% of the time, indicating there are many cases for which either the existing comment did not need updating, or comments produced by all models were poor. Based on our inspection of a sample these, we observe that in a large portion of these cases, the comment did not warrant an update. This is consistent with prior work in sentence simplification which shows that, very often, there are sentences that do not need to be simplified (Li and Nenkova, 2015). Despite our efforts to minimize such cases in our dataset through rule-based filtering techniques, we found that many remain. This suggests that it would be beneficial to first determine whether a comment needs to be updated before proposing a revision. We address this in Chapter 6 by integrating the inconsistency detection classifiers from Chapter 4 with the comment update model, to build a combined system which updates a comment only if it becomes inconsistent upon code changes.

Chapter 6

Combined Detection + Update of Inconsistent Comments

In Chapters 4 and 5, we explored the tasks of detecting inconsistent comments and updating them in isolation. We now combine models for these two tasks to build a comprehensive just-in-time comment maintenance system which first determines whether a comment, C_{old}, has become inconsistent upon code changes to the corresponding method ($M_{old} \rightarrow M_{new}$), and then automatically suggests a revision if this is the case. Full details of this work are available in Panthaplackel et al. (2021b).

6.1. Experiments

We use the dataset that we introduced in Section 4.3. Recall that *positive* examples correspond to cases in which both the method and comment are changed, and *negative* examples correspond to cases in which only the method is changed. We consider three different configurations for combining our inconsistency detection models (Section 4.4.2) with our comment update model (Section 5.2).

- **Update w/ implicit detection:** We augment training of the update model with negative examples in which C_{old} does not need to be updated. This baseline implicitly performs inconsistency detection by learning to copy C_{old} when an update is not needed. We evaluate with respect to inconsistency detection based on whether or not it predicts C_{old} as C_{new}.

- **Pretrained update + detection:** The update and detection models are trained separately. At test time, if the detection model classifies C_{old} as inconsistent, we take the prediction of the update model. Otherwise, we copy C_{old}, making C_{new}=C_{old}. We consider three of our just-in-time detection models.

- **Jointly trained update + detection:** We jointly train the inconsistency detection model with the update model on the full dataset (including positive and negative examples). We consider all three of our just-in-time detection techniques. The update model and detection model share embeddings and the comment encoder for all three, and for the sequence-based and hybrid models, the code sequence encoder is also shared. During training, loss is computed as the sum of the update and detection components. For negative examples (i.e., C_{old} does not need to be updated), we mask the loss of the update component since it does not have to learn

	Update Metrics				
	xMatch	METEOR	BLEU-4	SARI	GLEU
Never update	50.0	67.4	72.1	24.9	68.2
Update model (Chapter 5)	25.9	60.0	68.7	42.0*	67.4
Update w/ implicit detection	58.0	72.0	74.7	31.5	72.7
Pretrained update + detection					
Seq(C_{old}, M_{edit}) + features	**62.3**†	75.6*	77.0*	42.0*	76.2
Graph(C_{old}, T_{edit}) + features	59.4	74.9§	76.6†	**42.5**$^{\|}$	75.8*†
Hybrid(C_{old}, M_{edit}, T_{edit}) + features	**62.3**†	75.8$^{\dagger\|}$	**77.2**	42.3†	**76.4**
Jointly trained update + detection					
Seq(C_{old}, M_{edit}) + features	61.4*	**75.9**$^{\|}$	76.6†	42.4$^{\dagger\|}$	75.6†
Graph(C_{old}, T_{edit}) + features	60.8	75.1§	76.6†	41.8*	75.8*
Hybrid(C_{old}, M_{edit}, T_{edit}) + features	61.6*	75.6*†	76.9*	42.3†	75.9*

Table 6.1: Comparing performance on update between combined systems on the cleaned test sample. Scores for which the difference in performance is *not* statistically significant are shown with identical symbols.

	Detection Metrics			
	P	R	F1	Acc
Never update	0.0	0.0	0.0	50.0
Update model (Chapter 5)	54.0	**95.6**	69.0	57.1
Update w/ implicit detection	**100.0**	23.3	37.7	61.7
Pretrained update + detection				
Seq(C_{old}, M_{edit}) + features	91.3*	82.0§	86.4*	87.1§¶
Graph(C_{old}, T_{edit}) + features	85.8	87.1	86.4*	86.3†
Hybrid(C_{old}, M_{edit}, T_{edit}) + features	92.3	82.4§	87.1†	87.8$^{*\|}$
Jointly trained update + detection				
Seq(C_{old}, M_{edit}) + features	88.3†	86.2	87.2†	87.3$^{\S\|}$
Graph(C_{old}, T_{edit}) + features	88.3†	84.7*	86.4*	86.7†¶
Hybrid(C_{old}, M_{edit}, T_{edit}) + features	90.9*	84.9*	**87.8**	**88.2***

Table 6.2: Comparing performance on inconsistency detection between combined systems on the cleaned test sample. Scores for which the difference in performance is *not* statistically significant are shown with identical symbols.

to copy C_{old}. At test time, if the detection component predicts a negative label, we can directly copy C_{old} and otherwise take the prediction of the update model.

6.2. Results

In Tables 6.1 and 6.2, we compare performances of combined inconsistency detection and update systems on the cleaned test sample. As reference points, we also provide scores for a system which never updates (i.e., always copies C_{old} as C_{new}) and our comment update model, which is designed to always update (and only copy C_{old} if an invalid edit action sequence is generated).

Since our dataset is balanced, we can get 50% exact match by simply copying C_{old} (i.e., never updating). In fact, this can even beat our comment update model on xMatch, METEOR, BLEU-4, and SARI, and GLEU. This underlines the importance of first determining whether a comment needs to be updated, which can be addressed with the inconsistency detection component. On the majority of the update metrics, both of these underperform the other three approaches (Update w/ implicit detection, Pretrained update + detection, and Jointly trained update + detection). SARI is calculated by averaging N-gram F1 scores for edit operations (add, delete, and keep). So, it is not surprising that the Update w/ implicit detection baseline, which learns to copy, performs fewer edits, consequently underperforming on this metric. Because our comment update model is designed to *always* edit, it can perform well on this metric; however, the majority of our pretrained and jointly trained systems can beat this.

The Update w/ implicit detection baseline, which does not include an explicit inconsistency detection component, performs relatively well with respect to the update metrics, but it performs poorly on detection metrics. Here, we use generating C_{old} as the prediction for C_{new} as a proxy for detecting inconsistency. It achieves high precision, but it frequently copies C_{old} in cases in which it is inconsistent and should be updated, hence underperforming on recall. The pretrained and jointly trained approaches outperform this model by wide statistically significant margins across the majority of metrics, demonstrating the need for explicitly performing inconsistency detection.

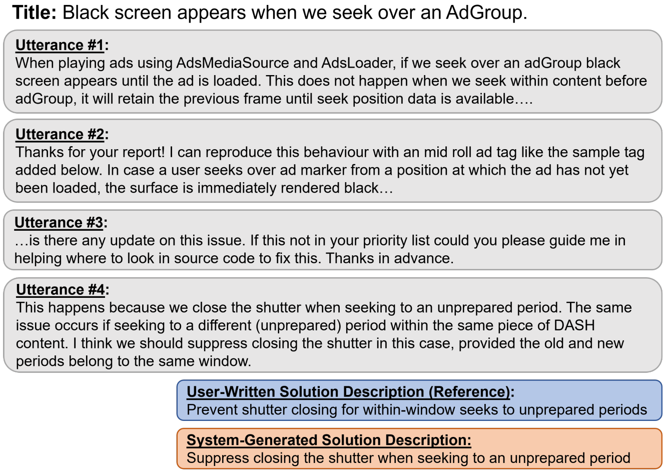
We do not observe a significant difference between the pretrained and jointly trained systems. The pretrained models achieve slightly higher scores on most update metrics and the jointly trained models achieve slightly higher scores on the detection metrics; however, these differences are small and often statistically insignificant. While we had expected the jointly trained system to perform better, neural networks are often overparameterized, so it is possible that a network can learn to fit both tasks, without having them affect one another.

Chapter 7

Describing Solutions for Bug Reports

In Chapters 4-6, we focus on detecting and updating natural language comments immediately *after* code changes to uphold software quality once these changes are merged into the code base. We now shift to using a different form of natural language, namely dialogue in bug report discussions, to instead quickly *drive* critical code changes for resolving bugs which threaten software quality. Bug report discussions can grow rapidly, through the many exchanges (Liu et al., 2020) among multiple participants (Kavaler et al., 2017), spanning several months or even longer (Kikas et al., 2015). The solution is often formulated within the discussion (Arya et al., 2019; Noyori et al., 2019); however, this can be challenging to locate and interpret amongst a large mass of text.

To enable developers to more easily absorb information relevant towards implementing the solution through the necessary code changes, we propose automatically generating a concise natural language description of the solution by synthesizing the relevant content as soon as it emerges in the discussion (Panthaplackel et al., 2021c). In this chapter, we focus on benchmarking models for the generation task. In the following chapter, we will introduce a secondary classification task for integrating it into a real-time setting to help quickly mobilize developers for implementation.



Title: Black screen appears when we seek over an AdGroup.

Utterance #1:
When playing ads using AdsMediaSource and AdsLoader, if we seek over an adGroup black screen appears until the ad is loaded. This does not happen when we seek within content before adGroup, it will retain the previous frame until seek position data is available….

Utterance #2:
Thanks for your report! I can reproduce this behaviour with an mid roll ad tag like the sample tag added below. In case a user seeks over ad marker from a position at which the ad has not yet been loaded, the surface is immediately rendered black…

Utterance #3:
…is there any update on this issue. If this not in your priority list could you please guide me in helping where to look in source code to fix this. Thanks in advance.

Utterance #4:
This happens because we close the shutter when seeking to an unprepared period. The same issue occurs if seeking to a different (unprepared) period within the same piece of DASH content. I think we should suppress closing the shutter in this case, provided the old and new periods belong to the same window.

User-Written Solution Description (Reference):
Prevent shutter closing for within-window seeks to unprepared periods

System-Generated Solution Description:
Suppress closing the shutter when seeking to an unprepared period

Figure 7.1: ExoPlayer[1] discussion with user-written and system-generated solution descriptions.

7.1. Task

As shown in Figure 7.1, when a user reports a bug, they state the problem in the *title* (e.g., "Black screen appears when we seek over an AdGroup") and initiate a discussion by making the first *utterance* (U_1), which usually elaborates on the problem (e.g., "When playing ads using AdsMediaSource and AdsLoader, if we seek over..."). Other participants

[1] `https://github.com/google/ExoPlayer/issues/5507`

join the discussion at later points in time through utterances ($U_2 ... U_T$), where T is the total number of utterances. Throughout the discussion, developers discuss various aspects of the bug, including a potential solution (Arya et al., 2019). As the discussion progresses, the cause of the bug is identified as the shutter getting closed "when seeking to an unprepared period" and a solution emerges: "suppress closing the shutter in this case, provided the old and new periods belong to the same window." We study the task of generating a concise description of the solution (e.g., "Prevent shutter closing for within-window seeks to unprepared periods") by synthesizing relevant content within the title and sequence of utterances ($U_1, U_2 ...$).

7.2. Data

We build a corpus by mining issue reports corresponding to open-source projects from GitHub Issues, as done in prior work (Kavaler et al., 2017; Panichella et al., 2021). We specifically collect examples from Java projects. Issue reports can entail feature requests as well as bug reports. In this work, we focus on the latter. We identify bug reports by searching for "bug" in the labels assigned to a report and by using a heuristic for identifying bug-related commits (Karampatsis and Sutton, 2020a).

7.2.1. Data Collection

A bug report is organized as an event timeline, recording activity from when the report is opened to when it is closed. From comments that are posted on this timeline, we extract utterances which form the *discussion* corresponding to a bug report, ordered based on their timestamps. We specifically consider bug reports that are linked to source code and documentation changes made in the code repository to resolve the bug (Nguyen et al., 2012). These changes are made through *commits* and *pull requests*, which also appear on the timeline. Changes made in a commit or pull request are described using natural language, in the corresponding commit message (Loyola et al., 2017; Xu et al., 2019a) or pull request title (Kononenko et al., 2018; Zhao et al., 2019) respectively. In practice, developers write commit messages and pull request titles after making code changes. However, much like prior work (Chakraborty and Ray, 2021), we treat them as a proxy for solution descriptions which can drive bug-resolving code changes.

Furthermore, we extract the position of a commit or pull request on the timeline, relative to the utterances in the discussion. We consider this as the point at which a developer acquired enough information about the solution to implement the necessary changes and describe these changes with the corresponding commit message or pull request title. So, if the implementation is done immediately after U_g on the timeline, then we take this position t_g as the "gold" time step for when sufficient context becomes available to generate an informative description of the solution. This leads to examples of the form *(Title, $U_1 ... U_T$, t_g, description)*. We disregard examples consisting of multiple commit messages and PR titles, so there is at most one example per issue report. However, for future work (Section 10.1), we believe such examples can be useful for to support generating descriptions at multiple time steps.

7.2.2. Handling Noise

Upon studying the data, we deemed it necessary to perform filtering for more effective supervision and accurate evaluation, as commonly done for tasks in this domain (Section 2.8). After applying simple heuristics to reduce noise, we obtain the examples which we focus on in this work, the *full dataset*. However, we identify three sources of noise that are more difficult to control with simple heuristics and propose techniques to quantify them. We use these to build a *filtered subset* of the full dataset that is less noisy. This subset is used for more detailed analysis of the models that are discussed in the paper, and we find that training on this subset leads to improved performance (Section 7.5).

- **Generic descriptions**: Commit messages and pull request titles are sometimes generic (e.g., *"fix issue."*) (Etemadi and Monperrus, 2020). To limit such cases, we compute normalized inverse word frequency (NIWF), which is used in prior work to quantify specificity (Zhang et al., 2018). The filter excludes 1,658 examples in which the reference description's NIWF score is below 0.116 (10th percentile computed from training data).

- **Uninformative descriptions**: Instead of describing the solution, the commit message or pull request title sometimes essentially re-state the problem (which is usually mentioned in the title of the bug report). To control for this, we compute the percentage of unique, non-stopword tokens in the reference description which also appear in the title. The filtered subset excludes 3,552 additional examples in which this percentage is 50% or more.

- **Discussions without sufficient context**: While enough context is available to a developer to implement a solution at t_g, this context may not always be available in the discussion and could instead be from their technical expertise or external resources. Sometimes, the solution is not mentioned within the discussion. For instance, in the discussion in the footnote[2], only a stack trace and personal exchanges between developers are present. From the utterance before the PR, "Or PM me the query that failed" suggests that an offline conversation occurred. Since relevant content is not available in such cases, it is unreasonable to expect to generate an informative description. We try to identify examples in which there is no useful content for generating the target output by using a previously proposed approach (Nallapati et al., 2017) for greedily constructing an extractive summary based on a reference abstractive summary. The filtered subset excludes 1,262 more examples for which a summary could not be constructed. After applying all three filtering techniques, we are left with 5,856 examples.

7.2.3. Preprocessing

We retain inlined code; however, we remove code blocks and embedded code snippets, as done in prior work (Tabassum et al., 2020; Ahmad et al., 2021). Capturing meaning from large bodies of code often requires reasoning with respect to the abstract syntax tree (Alon et al., 2019) and data and control flow graphs (Allamanis et al., 2018b). We also do not use source code files within a project's repository. We leave it to future work to incorporate large bodies of code. We discard URLs and mentions of GitHub usernames

[2] `https://github.com/prestodb/presto/issues/14567`

	Train	**Valid**	**Test**	**Total**
Projects	395 (330)	145 (111)	134 (104)	412 (344)
Examples	9,862 (4,664)	1,232 (599)	1,234 (593)	12,328 (5,856)
# Commit messages	4,520 (2,355)	410 (234)	386 (189)	5,316 (2,778)
# PR titles	5,342 (2,309)	822 (365)	848 (404)	7,012 (3,078)
Avg T	3.9 (4.5)	3.8 (4.4)	4.0 (4.4)	3.9 (4.5)
Avg t_g	2.9 (3.4)	2.9 (3.4)	3.2 (3.6)	2.9 (3.4)
Avg utterance length (#tokens)	68.4 (75.6)	74.8 (84.3)	70.2 (75.7)	69.2 (76.5)
Avg title length (#tokens)	10.6 (10.6)	11.2 (11.0)	11.5 (11.3)	10.7 (10.7)
Avg description length (#tokens)	9.1 (10.5)	8.9 (9.9)	9.1 (10.1)	9.1 (10.4)

Table 7.1: Data statistics. In parentheses, we show metrics computed on the filtered subset.

from utterances. From the description, we remove references to issue numbers and pull request numbers.

7.2.4. Partitioning

The dataset spans bug reports from April 2011 - July 2020. We partition the dataset based on the timestamp of the commit or pull request associated with a given example. Namely, we require all timestamps in the training set to precede those in the validation set and all timestamps in the validation set to precede those in the test set. Partitioning with respect to time ensures that we are not using models trained on future data to make predictions in the present, more closely resembling the real-world scenario (Nie et al., 2021). Dataset statistics are shown in Table 7.1.

7.3. Models

		1	**2**	**3**	**4**
Full	Title	73.0	88.9	94.0	96.1
	$U_1 ... U_{t_g}$	54.7	87.6	95.0	97.6
	Title + $U_1 ... U_{t_g}$	47.9	82.0	91.2	94.8
Filtered	Title	82.3	95.6	98.4	99.4
	$U_1 ... U_{t_g}$	49.9	87.4	95.1	97.8
	Title + $U_1 ... U_{t_g}$	47.5	86.0	94.5	97.5

Table 7.2: Percent of novel unigrams, bigrams, trigrams, and 4-grams in the reference description, with respect to the title, $U_1 ... U_{t_g}$, and title + $U_1 ... U_{t_g}$. The high percentages show that generating solutions is an abstractive task.

We benchmark various models for generating informative solution descriptions in a static setting, in which we leverage the oracle context from the discussion (i.e., the title and $U_1 ... U_{t_g}$). From Table 7.1, the average length of a single utterance is ∼70 tokens while the average description length is only ∼9 tokens. Therefore, this task requires not only effectively selecting content about the solution from the long context (which could span multiple utterances) but also synthesizing this content to produce a concise description. Following See et al. (2017), we compute the percent of novel n-grams in the reference description with respect to the input context in Table 7.2. The high percentages underline the need for an *abstractive* approach, rather than an *extractive* one which generates a description by merely copying over utterances or sentences within the discussion.[3] Furthermore, success

[3] We observe very low performance with extractive approaches.

on this task requires complex, bimodal reasoning over technical content in the discussion, encompassing both natural language and source code.

We describe the models we consider below. To represent the input in neural models, we insert <TITLE_START> before the title and <UTTERANCE_START> before each utterance.

- **Copy Title:** Though the bug report title typically only states a problem, we observe that it sometimes also puts forth a possible solution, so we evaluate how well it can serve as a concise description of the solution.

- **Seq2Seq + Ptr:** We consider a transformer encoder-decoder model in which we flatten the context into a single input sequence (Vaswani et al., 2017). Generating the output typically requires incorporating out-of-vocabulary tokens from the input that are specific to a given software project, so we support copying with a pointer generator network (Vinyals et al., 2015).

- **Hier Seq2Seq + Ptr:** Inspired by hierarchical approaches for dialogue response generation (Serban et al., 2016), we consider a hierarchical variant of the Seq2Seq + Ptr model with two separate encoders: one that learns a representation of an individual utterance, and one that learns a representation of the whole discussion. We encode U_t using a transformer-based encoder and feed the contextualized representation of its first token (<UTTERANCE_START>) into the RNN-based discussion encoder to update the *discussion state*, s_t. When encoding U_t, we also concatenate s_{t-1} to embeddings, to help the model relate U_t with the broader context of the discussion. Note that we treat the title as U_0 in the discussion. This process continues until U_{t_g} is encoded, at which point all accumulated token-level hidden states are fed into a transformer-based decoder to generate the output. Unlike the Seq2Seq + Ptr model which is designed to reason about the full input at once, this approach reasons step-by-step, with self-attention in the utterance encoder only being applied to tokens within the same utterance. Since the input context for this task is often very large, we investigate whether it is useful to break down the encoding process in this way. We also equip this model with a pointer generator network.

- **PLBART:** Ahmad et al. (2021) recently proposed PLBART, which is pretrained on a large amount of code from GitHub and software-related natural language from StackOverflow, using BART-like (Lewis et al., 2020) training objectives. With fine-tuning, PLBART achieves state-of-the-art performance on many program and language understanding tasks like code summarization/generation. We fine-tune PLBART on our training set and evaluate its ability to comprehend bug report discussions and generate descriptions of solutions.[4] Note that PLBART truncates input to 1024 tokens.

- **PLBART (F):** Since PLBART is pretrained on a large amount of data, we can afford to reduce the fine-tuning data. So we fine-tune on only the filtered subset of the training set (Section 7.2.2), to investigate whether fine-tuning on this "less noisy" sample can lead to improved performance.

[4] We use PLBART rather than vanilla BART because it achieves higher performance for our task.

	Model	BLEU-4	METEOR	ROUGE-L
Full	Copy Title	$14.4^{\|}$	13.1	$24.4^{\S}$
	Seq2Seq + Ptr	12.6	9.8	$25.0^{\ddagger}$
	Hier Seq2Seq + Ptr	12.4	9.6	$24.1^{\S}$
	PLBART	**16.6**	**14.5**	**28.3**
	PLBART (F)	$14.2^{\|}$	12.3	$25.1^{\ddagger}$
Filtered	Copy Title	$10.0^{*\dagger}$	8.3	16.6
	Seq2Seq + Ptr	10.2^{*}	7.5	20.1
	Hier Seq2Seq + Ptr	$9.9^{\dagger}$	7.4	19.6
	PLBART	$\mathbf{12.3}^{\ddagger}$	9.9	21.1
	PLBART (F)	$\mathbf{12.3}^{\ddagger}$	**10.2**	**21.9**

Table 7.3: Automated metrics for generation. Scores for Seq2Seq + Ptr and Hier Seq2Seq + Ptr are averaged across three trials. Differences that are *not* statistically significant are indicated with matching symbols.

7.4. Results: Automated Metrics

We compute common text generation metrics, BLEU-4 (Papineni et al., 2002), METEOR (Banerjee and Lavie, 2005), and ROUGE-L (Lin, 2004). We compute statistical significance with bootstrap tests (Berg-Kirkpatrick et al., 2012) with $p < 0.05$. Results are in Table 7.3. On the full test set, PLBART outperforms other models by statistically significant margins, demonstrating the value of pretraining on large amounts of data[5]. PLBART (F) underperforms PLBART on the full test set; however, on the filtered subset, PLBART (F) either beats or matches PLBART. We find that there is a large drop in performance across models between the full test set and filtered subset. As demonstrated by the relatively high performance of the naive Copy Title baseline, models can perform well by simply copying or rephrasing the title in many cases, for the full test. However, the filtered subset is designed to remove uninformative reference descriptions that merely re-state the problem. Nonetheless, because critical keywords relevant to the solution are often also in the title, the Copy Title baseline can still achieve reasonable scores on the filtered subset, even beating Seq2Seq + Ptr and Hier Seq2Seq + Ptr on METEOR. Although automated metrics provide some signal, they emphasize syntactic similarity over semantic similarity. For further evaluation, we conduct human evaluation.

7.5. Results: Human Evaluation

Users are asked to read through the content in the title and the discussion ($U_1 ... U_{t_g}$). For each example, they are shown predictions from the 5 models discussed in Section 7.3, and they must select one or more of the descriptions that is most informative towards resolving the bug. If all candidates are uninformative, then they select a separate option: "All candidates are poor." There is also another option to indicate that there is insufficient context about the solution (Section 7.2.2), making it difficult to evaluate candidate descriptions. They must also write a rationale for their selection. Before starting the annotation task, users must watch a training video in which we walk through seven

[5] While Seq2Seq + Ptr and Hier Seq2Seq + Ptr are slightly smaller than PLBART in model size, we find that randomly initializing a model resembling PLBART's architecture results in lower performance than both of these.

examples in detail.

Since annotation requires not only technical expertise, but also high cognitive load and time commitment, it is hard to perform human evaluation on a large number of examples with multiple judgments per example. Similar to Iyer et al. (2016), we resort to having each example annotated by one user to obtain more examples. We recruited 8 graduate students with 3+ years of programming experience and familiarity with Java. Each user annotated 20 examples, leading to annotations for 160 unique examples in the full test set. Note that these users are not active contributors, thus they will likely select the option pertaining to insufficient context more often than if they were active contributors to these projects who have a deeper understanding of their implementations. However, it is difficult to conduct a user study at a similar scale with contributors. Nonetheless, there are developers aiming to become first-time contributors for a particular project (Tan et al., 2020). Our study better aligns with this use case.

In Table 7.4, we show that PLBART (F) substantially outperforms all other models, with users selecting its output 33.1% of the time. Even though the title typically only states a problem, users selected it 8.1% of the time. From rationales that users were asked to write, we found that there were cases in which the title not only posed the problem but also offered a solution. Users rarely preferred the output of Seq2Seq + Ptr and Hier Seq2Seq + Ptr as they usually just rephrased the problem. PLBART also appears to be re-stating the problem in many cases; however, less often than other models.

Though we see similar trends across the full test set and the filtered subset, all models except PLBART (F) tend to perform worse on the filtered subset, as previously observed on automated metrics. Also, the average number of cases with insufficient context is lower for the filtered subset, confirming that we are able to reduce such cases through filtering. We find the results on the filtered data to align better with human judgment. By fine-tuning on the filtered training set, PLBART (F) learns to pick out important information from within the context and generate descriptions which reflect the solution rather than the problem.

Model	**Full**	**Filtered**
Copy Title	8.1	6.0
Seq2Seq + Ptr	1.3^*	$1.2^\dagger$
Hier Seq2Seq + Ptr	1.3^*	$1.2^\dagger$
PLBART	11.9	10.5
PLBART (F)	**33.1**‡	**39.5**
All Poor	20.0	22.1
Insufficient Context	$31.9^\ddagger$	25.6

Table 7.4: Human evaluation results: Percent of annotations for which users selected predictions made by each model. This entails 160 annotations for the full test set, 86 of which correspond to examples in our filtered subset. Differences that are *not* significant are indicated with matching symbols.

7.5.1. Analysis

In Table 7.5, we show model outputs for the example in Figure 7.1. Seq2Seq + Ptr and Hier Seq2Seq + Ptr essentially rephrase aspects of the problem, which are described in the title. Both PLBART and PLBART (F) capture the solution, with PLBART (F) providing more information. When there is sufficient context, 62.4% of the time, either PLBART or PLBART (F) generates output that is informative towards bug resolution. While this demonstrates that fine-tuning this

Model	**Prediction**
Copy Title	black screen appears when we seek over an ad group .
Seq2Seq + Ptr	fix black ads
Hier Seq2Seq + Ptr	fix seeking in ad tag
PLBART	suppress closing shutter when seeking over an ad group
PLBART (F)	suppress closing the shutter when seeking to an unprepared period
Reference	prevent shutter closing for within - window seeks to unprepared periods

Table 7.5: Model outputs for the example

	Title	**PLBART (F)**	**Reference**
(1)	Issue with dex: OIDC server is not available at the 'quarkus.oidc.auth-server-url' URL	fix trailing slash in auth - server url	strip trailing forward slash from oidc url
(2)	InvalidDataTypeException: UDATA contains value larger than Integer.MAX_VALUE DDR issue decoding lookswitch	fix bug in byte code dumper when tableswitch instruction precedes tableswitch instruction	fix interpretation of switch instructions in byte code dumper
(3)	Worldmap viewport changes when switching between dashboard pages	don ' t refresh widget grid when worldmap loses viewport	define key prop for map visualization to update map on dimension change
(4)	Workaround comments exist in opengrok-indexer/pom.xml file while the related issues are already fixed.	fix jflex - de / jflex # 705 (comment)	use jflex 1.8.2
(5)	Why subscribe with single action for onNext design to crush if error happened?	1 . x : fix subscription . subscribe ( ) to return observable . empty ( ) 2 . x : fix subscription . subscribe ( ) to return observable . empty ( )	fixed sonar findings

Table 7.6: Output of PLBART (F) for a sample of examples in the test set:
(1) `https://github.com/quarkusio/quarkus/issues/10227`,
(2) `https://github.com/eclipse-openj9/openj9/issues/9294`,
(3) `https://github.com/Graylog2/graylog2-server/issues/7997`,
(4) `https://github.com//oracle/opengrok/issues/3172`,
(5) `https://github.com/ReactiveX/RxJava/issues/637`.

large, pretrained model on our data can be useful in supporting bug resolution in on-line discussions to some extent, it also shows that there is opportunity for improvement.

We manually inspected PLBART (F)'s outputs and associated user rationales. We observe that the model tends to perform better when the solution is clearly stated in 1-3 consecutive sentences (Table 7.6 (1) and (2)). When more complex synthesis is needed, it sometimes stitches together tokens from the input incorrectly (Table 7.6 (3)). Next, although the model picks up on information in the context, sometimes, it draws content from an elaboration of the problem from within the discussion rather than a formulation of the solution (Table 7.6 (4)). This demonstrates that it still struggles to disentangle content relevant to the solution from that about the problem. We also find that it sometimes struggles to generate meaningful output when in-lined code is present, highlighting the challenge in bimodal reasoning about code and natural language (Table 7.6 (5)). Finally, we find problems with repetition and fluency (Table 7.6 (1)), as commonly seen in the outputs of neural models (Holtzman et al., 2020).

Chapter 8

Describing Solutions for Bug Reports in Real-Time

For the generated solution descriptions from Chapter 7 to be useful in resolving bugs, generation must be performed during ongoing discussions. In a real-time setting, the formulation of the solution is likely not immediately available but rather emerges as the discussion progresses and the sequence of utterances grows. So, we propose an additional task for monitoring progress in an ongoing discussion to predict the time step t in which the title and $U_1 ... U_t$ constitute sufficient context for generating an informative description. For this, we train a binary classifier to predict the time step (t_g) in which the necessary context is available, and we combine it with the generation task as a preliminary investigation for a real-time generation system. More concretely, in Figure 7.1, the solution is formulated in U_4, so the correct behavior is for the classifier to predict the negative label at $t = 1, 2, 3$ and the positive label at $t = 4$. Once the positive label is predicted, the description is generated, conditioned on the title and $U_1 ... U_{t_p}$.

8.1. Our Classifier

Our approach sequentially processes each new utterance and decides if it adds enough information to propose a solution. We first prepend <TITLE_START> to the sequence of tokens in the title and encode it with a transformer-based encoder. We consider the contextualized representation of this token as a vector representation of the information available at t_0, which we denote as r_0. Next, to process an utterance U_t ($t > 0$), we prepend <UTTERANCE_START> to the sequence of utterance tokens. We concatenate the representation at the previous time step (r_{t-1}) to the token embeddings and pass them through the encoder. The contextualized representation of the special token becomes r_t. Finally, we pass r_t through 3 linear layers and a sigmoid layer, and then apply softmax to classify whether or not a solution can be formulated at step t. By feeding in r_{t-1}, we inform the model of the prior context and evaluate the information added by U_t. We weight the positive and negative labels using the inverse of the class proportion to handle class imbalance (1.543 and 0.740 respectively). Additionally, to improve learning, we augment the training data with 12,350 non-bug examples, but apply a lower weight for these examples (0.7). The model is trained to minimize cross entropy loss.

		First	**Second**	**Rand (uni)**	**Rand (dist)**	**RF**	**Ours**
Full	Overall	29.5	26.0	23.3	24.6	21.9	**32.5**
	$t_g = 1$	**100.0**	0.0	49.4	52.7	30.9	62.9
	$t_g = 2$	0.0	**100.0**	24.7	26.3	35.8	33.4
	$t_g = 3$	0.0	0.0	10.6	10.0	11.5	**16.4**
	$t_g = 4$	0.0	0.0	5.2	4.7	8.7	**15.8**
	$t_g \geq 5$	0.0	0.0	0.9	1.3	4.3	**5.6**
Filtered	Overall	23.8	24.8	21.4	21.4	19.1	**28.8**
	$t_g = 1$	**100.0**	0.0	52.0	53.0	23.4	57.2
	$t_g = 2$	0.0	**100.0**	26.8	23.8	39.2	34.2
	$t_g = 3$	0.0	0.0	9.5	12.1	12.5	**19.4**
	$t_g = 4$	0.0	0.0	4.6	5.0	7.5	**15.8**
	$t_g \geq 5$	0.0	0.0	1.2	1.7	3.7	**7.0**

Table 8.1: Accuracy (i.e., percent of times $t_p = t_g$) overall and for varying t_g. All differences are statistically significant.

8.2. Classification Baselines

To better evaluate our classifier for determining when sufficient context is available for generating an informative description, we introduce some simple baselines. We observe that there are many cases in which $t_g = 1, 2$, i.e., the solution is implemented immediately after the first or second utterance. So, we include the FIRST baseline which always predicts a positive label at $t = 1$, and SECOND which predicts negative at $t = 1$ and positive at $t = 2$, if $t_g \geq 2$ (otherwise it never predicts positive).

Next, we include the RAND (uni) baseline which progresses through the discussion, randomly deciding between the positive and negative label after each utterance, based on a uniform distribution. We additionally include RAND (dist), which instead uses the probability distribution of labels at the example-level estimated from the training and augmentation data (i.e., pos = $\frac{1}{N}\sum_{n=1}^{N}\frac{1}{t_g}$=0.549, neg = 0.451).

Finally, we include a random forest classifier (RF) which makes a classification following each utterance, U_t, until the positive label is predicted or $t > t_g$. It uses TF-IDF representations of the title and U_t as well as an aggregated representation of $U_1 ... U_t$. Additionally, it uses the following features: the position t, length of U_t, author of U_t (as an index, with ordering dependent on entry into the discussion), frequency of utterances made by the author, the ratio of the length of U_t to the accumulated length of $U_1 ... U_t$, and the title length.

8.3. Classification Results

We evaluate on the full and filtered test sets from Section 7.2. We present results in Table 8.1. Results for the random baselines, random forest classifier, and our classifier are averaged across 3 trials. On both test sets, our classifier achieves the highest overall accuracy than. For longer discussions (with higher values of t_g), we observe that RF and our classifier, which dynamically make content-driven predictions, manage to outperform other baselines. In general, our classifier still outperforms RF, which we attribute to the more complex transformer-based architecture yielding better utterance representations. We find that that accuracy deteriorates substantially as t_g increases, illustrating the challenge in handling long dialogues.

The classifier fails to predict the positive label (before or at t_g) in some cases ($t_p =$ None). On the examples that it does predict the positive label, on average, t_p is 1.704, 1.895, and 1.804 time steps before t_g for the full, filtered, and curated test sets respectively. While a model should wait until sufficient context is available before generating, sometimes, the last couple utterances before the implementation do not add context about the solution but are rather personal exchanges between developers (e.g., "Thanks for the bug report", "I'll open a PR"). For this reason, we believe that predicting the positive label slightly before t_g is acceptable in certain cases.

8.4. Combined System

Finally, we combine the classifier and PLBART (F) (the best generation model from human evaluation) to build a complete system for deciding when a solution can be proposed and then generating one. In Table 8.2, we report automated metrics for PLBART (F), comparing model output between using the context up till t_p (the predicted time step of classifier) versus t_g. We observe that across metrics, predictions generated by the same underlying model using the context at t_g achieve higher scores than those made using the context at t_p. This highlights the gap in performance caused by error propagation from the classifier. We plan to investigate a higher-performing classifier (Section 9.1) and more intricate end system that is jointly trained on generation and classification (Section 9.2) in the future.

		BLEU-4	**METEOR**	**ROUGE-L**
Full	@t_p	11.3	9.9	19.9
	@t_g	**14.2**	**12.3**	**25.1**
Filtered	@t_p	9.5	7.8	16.3
	@t_g	**12.3**	**10.2**	**21.8**

Table 8.2: Comparing PLBART (F)'s performance with context available at t_p vs. t_g. If $t_p =$ None (i.e., positive label is not predicted before or at t_g), the predicted description is the empty string. All differences are statistically significant.

Chapter 9

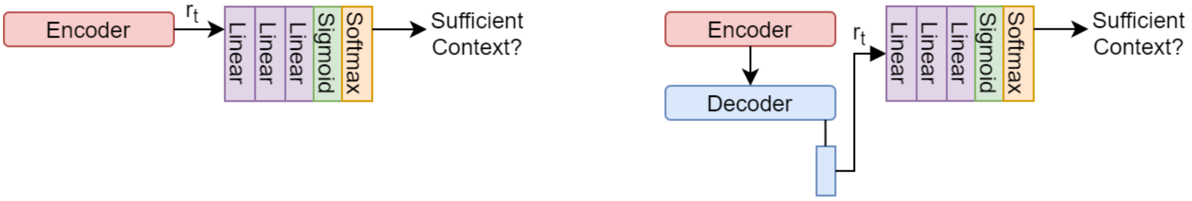
Proposed Work

For our first short-term goal, we aim to improve the classifier from Section 8.1 that determines when sufficient context for generating an informative description emerges in an ongoing discussion. The second short-term goal revolves around building an improved combined system (relative to the pipelined approach in Section 8.3) that is jointly trained on generation and classification.

9.1. Improving Classifier

In Chapter 8, we conducted an initial study of a real-time system which generates solution descriptions during an ongoing discussion, by pipelining a generation model with a classifier which determines when to perform generation. Using our classifier's predicted label instead of the oracle label yielded much lower performance. Therefore, to improve overall performance, we need a higher-performing classifier. In Section 7.4, we found that using a pretrained model substantially outperforms one that is randomly initialized for the generation task, which operates on the same type of input as the classification task. As shown in Figure 9.1a, our classifier entails a transformer encoder, identical to the encoder used for Seq2Seq + Ptr and Hier Seq2Seq + Ptr, followed by 3 linear layers, a sigmoid layer, and a final softmax layer (Section 8.1). We intend to conduct experiments in which we replace the randomly initialized transformer encoder with one that is pretrained. For this purpose, we plan to explore CodeBERT (Feng et al., 2020) (trained on code and comments from GitHub), BERTOverflow (Tabassum et al., 2020) (trained on technical text from StackOverflow), and finally PLBART (trained on code from GitHub and technical text from StackOverflow). For PLBART, we intend to conduct two separate experiments. The first is simply using its encoder in the same way as before. Additionally, since PLBART is trained as a denoising autoencoder, we could also leverage the decoder. As shown in Figure 9.1b, we will add a special token to the *end* of the input sequence and take the final hidden state of the decoder corresponding to this token as the sequence representation, similar to how BART is applied to sequence classification tasks (Lewis et al., 2020).

To provide the model with broader context of the discussion, we had concatenated the encoder's learned representation for the previous utterance, r_{t-1}, to the input embeddings when encoding the current utterance, U_t. We plan to investigate an alternative strategy for incorporating this context, in which we feed in an aggregated sequence that closely resembles the structure of the input into PLBART for the generation task: (*title*, $U_1, U_2, ..U_t$). Through the self-attention layers (Vaswani et al., 2017), we believe the en-



(a) Basic architecture of our classifier from Section 8.1.

(b) Classifier using PLBART in which the decoder's final hidden state for the last token is passed into the classification layers.

Figure 9.1: Architectures for the the classification task of determining whether there is sufficient context to generate an informative description.

coder will better learn how U_t's content relates to the broader context of the discussion so far. Moreover, because the formulation of the solution is not always contained within a single utterance, this input format allows the encoder to better reason about whether sufficient context about the solution has accumulated across utterances at time step t. To account for PLBART's 1024 token limit, we will remove overflowing tokens from the *left side* of the subsequence of tokens corresponding to $U_1, ...U_{t-1}$. In addition to preserving critical tokens from the title and U_t, this also prioritizes tokens derived from more recent utterances in the discussion.

Additionally, by feeding in the representation of the previous utterance, we are implicitly capturing the ordering of utterances within the discussion. To *explicitly* specify ordering[1] and provide other salient information, we plan to incorporate additional features by concatenating a feature vector to r_t before we pass it through the classification layers. These features include the following: position of t, author of U_t (as an index, with ordering dependent on entry into the discussion), frequency of utterances made by the author, and the ratio of the length of U_t to the accumulated length of $U_1 ... U_t$. We found these features to be useful during experiments with a random forest baseline classifier, especially for longer discussions.

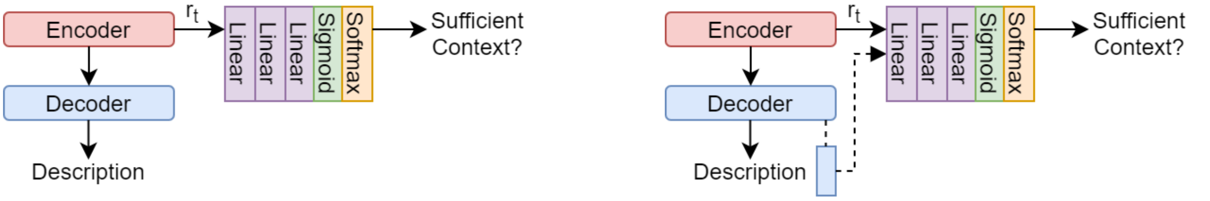
9.2. Jointly Training on Generation and Classification

Up till now, we have treated generating solution descriptions and classifying when to perform generation as independent tasks; however, they are inherently intertwined. It is not possible to generate an informative description without sufficient context, and "sufficient" context is defined by whether it can be used to generate an informative description. To allow these tasks to better complement one another, we intend to build an end system which is jointly trained on both generation and classification.

We provide an overview of our proposed architecture in Figure 9.2a. The two tasks will share an encoder, for which the input will be (*title*, $U_1, U_2, ...U_t$). There will be a separate decoder for the generation task and a separate set of layers for classification. We will initialize the encoder and decoder from PLBART. Conceptually, this is very similar to the Jointly trained update + detection model from Section 6.1.

We will compute the loss as the sum of the losses associated with the two individual tasks. Because the classifier runs after each new utterance, the classifier should be

[1] Using positional encodings (Vaswani et al., 2017) to represent utterance ordering did perform well in our initial experiments.



(a) Basic architecture for multi-task learning with a shared encoder and a separate decoder for generation and layers for classification.

(b) Variation of the architecture in Figure 9.2a in which the final decoder state from generation guides the classification task by capturing the notion of an *informative description.*

Figure 9.2: Architecture for jointly learning the generation and classification tasks needed for a real-time system that generates solution descriptions by synthesizing the relevant content in the discussion as it emerges.

trained at g time steps for any given example of the form ($title$, $U_1, U_2, ...U_g$). However, the generation model should be trained to only generate at one time step, t_g, when sufficient context for generating an informative description is available. To account for this discrepancy, we will mask the generation loss for $t < t_g$ during training. During inference, if the model predicts the positive label at t_p, then we take the generated description at t_p as the final prediction for the solution description.

We believe the classification task can guide content selection for the generation task by forcing the model to identify the specific parts of the input which contribute to predicting the positive label, i.e., the "sufficient context." Because we frame the classification task as determining whether sufficient context for generating an *informative description* is available, we a slight variation in which we leverage learned representations from the generation task to demonstrate what qualifies as an *informative description*. Namely, we will feed the final decoder hidden state corresponding to the last token as an additional input into the classification layers, as shown in Figure 9.2b. In principle, the decoder will not generate an informative description before t_g, so we believe this additional input will provide some signal that will be useful in learning to predict the negative label at $t < t_g$ and the positive label at t_g.

We will compare the performances of these combined systems with the pipelined system described in Section 8.4 as well as a new pipelined system using the best classifier from Section 9.1. We will also compare the quality of the generated output at t_p and t_g, to assess the impact of error propagation from classification. For evaluation, we plan to again use automated metrics (Section 7.4) and human evaluation (Section 7.5).

Chapter 10

Bonus Contributions

In this chapter, we discuss topics of interest for future work, which may be presented in the final dissertation.

10.1. Interactively Generating Descriptions to Drive Code Changes

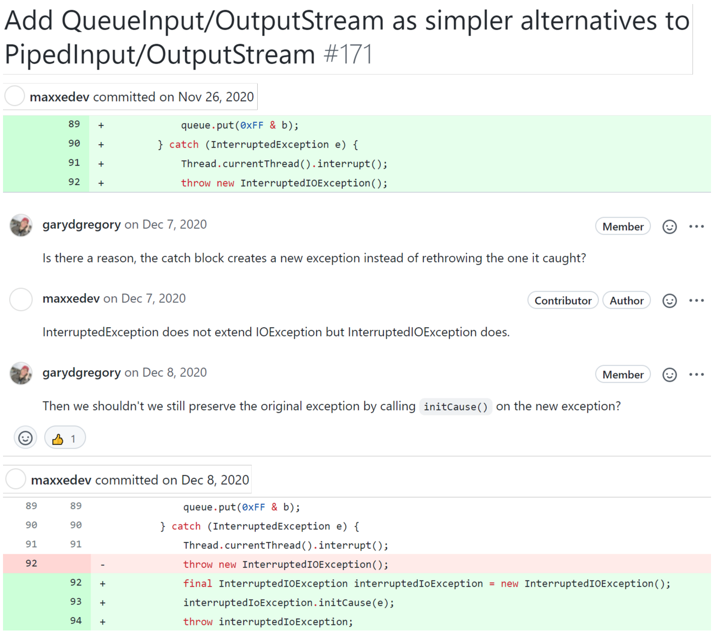
A system that generates an NL description of the solution when sufficient context emerges in an ongoing bug report discussion acts as an intelligent agent which chimes in to facilitate the implementation of the necessary code changes. However, this system is not interactive (i.e., it cannot react to new utterances made after it performs generation). We are interested in interactively generating descriptions to guide developers in making code changes.

10.1.1. Interacting in Bug Report Discussions

The dataset we consider in Chapters 7-9 consists of 12,328 bug report discussions linked to a single set of code changes (a single commit or PR), and thus there is only one associated NL solution description. We had also mined 4,571 bug-related and 12,609 nonbug-related issue reports corresponding to multiple commits and PRs, which we could use for learning to perform generation at multiple time steps. For instance, if there are commits at t_g and t_{g+k}, the system should first determine that sufficient context for generating a solution description is available at t_g and generate $\textit{description}_{\mathrm{g}}$ using ($\textit{title}$, $U_1, ...U_g$) as context. Then, it should continue monitoring progress in the discussion. At t_{g+k} it should recognize that another set of code changes is required, so it would generate another solution description, $\textit{description}_{\mathrm{g+k}}$, using ($\textit{title}$, $U_1, ...U_{g+k}$, $\textit{description}_{\mathrm{g}}$) as context. It would continue monitoring progress until the discussion terminates. For this, we plan to first study the data we mined more closely in order to understand the nature of the code changes in follow-up commits and PRs (e.g., the solution could have been implemented in parts or the first solution may have been incorrect and it is later corrected).

10.1.2. Interacting in Code Review Discussions

Next, we plan to study the task of interactively generating NL descriptions to guide code changes in a slightly different domain: code reviewing. Upon making modifications within



Add QueueInput/OutputStream as simpler alternatives to PipedInput/OutputStream #171

maxxedev committed on Nov 26, 2020

```
+            queue.put(0xFF & b);
+        } catch (InterruptedException e) {
+            Thread.currentThread().interrupt();
+            throw new InterruptedIOException();
```

garydgregory on Dec 7, 2020 — Member

Is there a reason, the catch block creates a new exception instead of rethrowing the one it caught?

maxxedev on Dec 7, 2020 — Contributor — Author

InterruptedException does not extend IOException but InterruptedIOException does.

garydgregory on Dec 8, 2020 — Member

Then we shouldn't we still preserve the original exception by calling `initCause()` on the new exception?

👍 1

maxxedev committed on Dec 8, 2020

```
             queue.put(0xFF & b);
         } catch (InterruptedException e) {
             Thread.currentThread().interrupt();
-            throw new InterruptedIOException();
+            final InterruptedIOException interruptedIoException = new InterruptedIOException();
+            interruptedIoException.initCause(e);
+            throw interruptedIoException;
```

Figure 10.1: PR discussion from `https://github.com/apache/commons-io/pull/171`.

a software project, a developer opens a PR so that other collaborators can review these modifications to evaluate whether they efficiently implement the correct functionality and adhere to established style guidelines (Brown and Parnin, 2020; Li et al., 2017). During this process, *reviewers* post review comments at specific locations in the code diff to point out problems they see and describe additional code changes for addressing these problems. The developer who opened the PR, or the *author*, then responds by posting comments or implementing the recommended changes. Reviewers may then post new comments in response, either addressing the author's comments or providing more feedback about the new changes (Tufano et al., 2021; Li et al., 2017). This can go on for a series of exchanges (Tsay et al., 2014; Golzadeh et al., 2019). For example, in Figure 10.1, maxxedev (the author) opens a PR for adding "QueueInput/OutputStream as simpler alternatives to PipedInput/OutputStream." The reviewer, garydgregory, comments on a *diff chunk* which introduces a catch block, to question why a new exception is being instantiated rather than simply rethrowing the original one. The author then responds to this by stating their rationale about a limitation with the InterruptedException class, for which the reviewer describes a possible workaround in the next comment: "preserve the original exception by calling `initCause`() on the new exception." The author then proceeds to implement this through code changes.

Because of the extensive manual effort that is required, reviewing can be very time-consuming (Hellendoorn et al., 2021; Jiang et al., 2021a; Wessel et al., 2020) and can also delay the release of critical software updates due to overloaded developers failing to complete reviewing in a timely manner (Yu et al., 2015; Maddila et al., 2020). Recently, there have been efforts to build tools for streamlining reviewing by automatically recommending relevant reviewers (Yu et al., 2014), PR prioritization (van der Veen et al., 2015), highlighting locations in the code diff which likely need a PR review comment (Hellendoorn et al., 2021) and providing a preview of how PR review comments should manifest as code changes (Tufano et al., 2021). We are interested in studying the prospects for an intelligent agent which can act as a reviewer to interactively suggest PR review comments to guide code changes that should be made during code review.

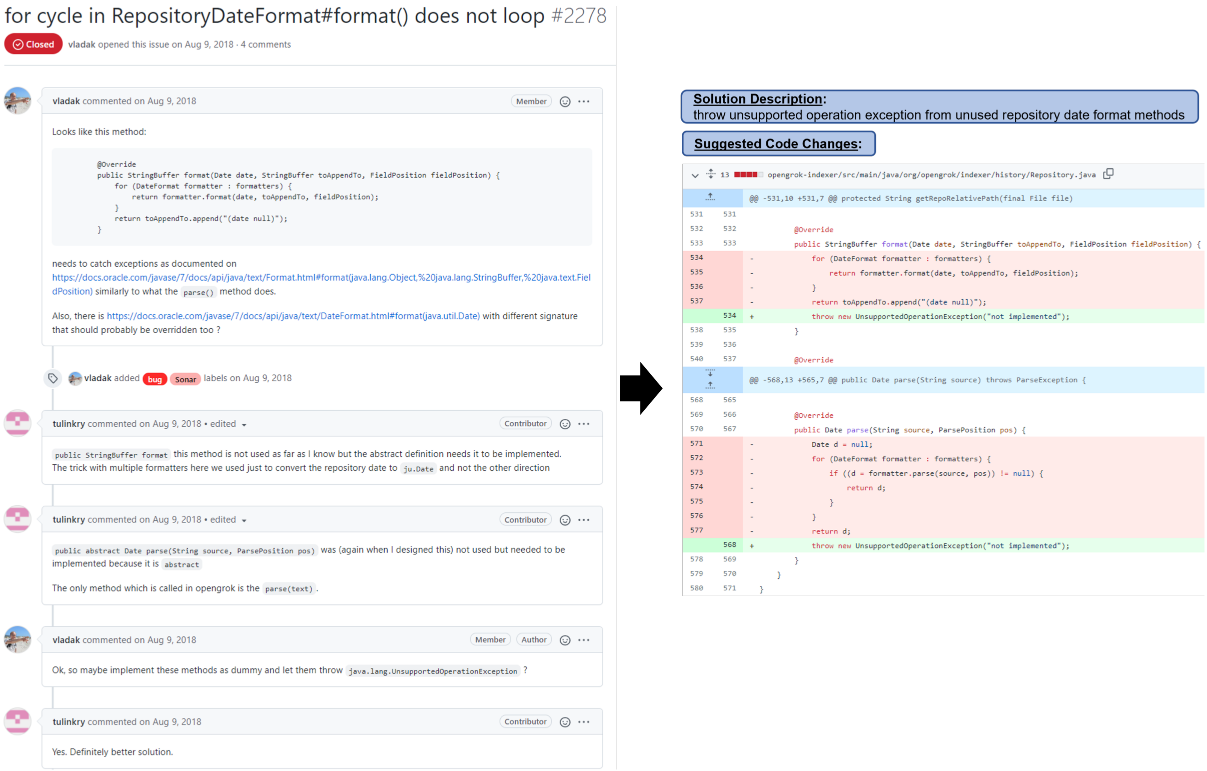
For a given diff chunk that is under question, we will model the dialogue between the author and the reviewer. The agent will assume the role of a reviewer. Note that multiple reviewers can be involved in an exchange with the author; however, we intend to treat this as a dyadic conversation in which we collapse all reviewers into a single role. As a first step, we assume the relevant diff chunk is specified, either manually or through automated tools (Hellendoorn et al., 2021). We treat the original diff chunk as the first utterance made by the author, A_1. Then, using A_1 as context, our agent will aim to generate a PR comment, R_1. The author will then either post a comment or make new code changes in response, A_2, and the agent will use A_1, R_1, A_2 as context to generate R_2. This process will continue until the agent determines that the final set of changes in the given diff chunk are acceptable.

We acknowledge that this is a very challenging task. While a model can learn to exploit common patterns in review comments, this task still requires complex technical reasoning, especially as the discussion progresses. So, we do not consider this as a standalone automated system but rather one that works alongside human reviewers. Namely, we envision a system that acts as a *first-pass reviewer* which generates comments early on in the discussion. Eventually, a human reviewer will have to intervene when the discussion reaches a stage which requires more technical expertise to comprehend the author's comments or code changes and respond in a meaningful way.

Here, any utterance made by the author or reviewer can consist of technical language or code. Since PLBART is trained to reason about code and technical language and also generate code and language, we plan to use this as the underlying architecture again. For data, we plan to first study the PR interactions associated with the 267,216 PRs released by (Tufano et al., 2019a). These PRs were mined from Gerrit[1], a platform for code review. Tufano et al. (2019a) specifically focus on the following three repositories: Android, Google, and Ovirt. From an initial inspection of this data, we found 63,734 PRs to have at least one interaction, resulting in a total of 287,648 interactions. For each of these PRs, there is on average 4.5 interactions, with each interaction entailing on average 2.01 comments (between authors and reviewers) that are attached to 1.44 lines of code. The interactions span on average 1.71 commits. Note that across commits, interactions could pertain to the same lines of code (e.g., the author may have made code changes which the reviewer has additional comments about). However, for this preliminary study, we did not merge interactions across commits as we will need to design an approach for performing this mapping.

We also plan to study this task for GitHub PRs as well. For an initial analysis, we considered the 21,778 PRs which are linked to the GitHub issue reports that we mined. From these, there are a total of 7,114 PRs with at least 1 interaction, resulting in a total of 25,625 interactions. For each of these PRs, there is on average 3.60 interactions, spanning 1.53 commits, with each interaction entailing 1.91 comments (between authors and reviewers) that are attached to 1.09 lines of code. Recall that these PRs were mined with the constraint that they had to be linked to a bug report. We plan to collect more PRs from GitHub by relaxing that constraint and also expanding to more projects.

[1] https://www.gerritcodereview.com



for cycle in RepositoryDateFormat#format() does not loop #2278

Closed vladak opened this issue on Aug 9, 2018 · 4 comments

vladak commented on Aug 9, 2018 — Member

Looks like this method:

```
        @Override
        public StringBuffer format(Date date, StringBuffer toAppendTo, FieldPosition fieldPosition) {
            for (DateFormat formatter : formatters) {
                return formatter.format(date, toAppendTo, fieldPosition);
            }
            return toAppendTo.append("(date null)");
        }
```

needs to catch exceptions as documented on https://docs.oracle.com/javase/7/docs/api/java/text/Format.html#format(java.lang.Object,%20java.lang.StringBuffer,%20java.text.FieldPosition) similarly to what the `parse()` method does.

Also, there is https://docs.oracle.com/javase/7/docs/api/java/text/DateFormat.html#format(java.util.Date) with different signature that should probably be overridden too ?

vladak added bug Sonar labels on Aug 9, 2018

tulinkry commented on Aug 9, 2018 • edited — Contributor

`public StringBuffer format` this method is not used as far as I know but the abstract definition needs it to be implemented. The trick with multiple formatters here we used just to convert the repository date to `ju.Date` and not the other direction

tulinkry commented on Aug 9, 2018 • edited — Contributor

`public abstract Date parse(String source, ParsePosition pos)` was (again when I designed this) not used but needed to be implemented because it is `abstract`

The only method which is called in opengrok is the `parse(text)`.

vladak commented on Aug 9, 2018 — Member Author

Ok, so maybe implement these methods as dummy and let them throw `java.lang.UnsupportedOperationException` ?

tulinkry commented on Aug 9, 2018 — Contributor

Yes. Definitely better solution.

Solution Description:
throw unsupported operation exception from unused repository date format methods

Suggested Code Changes:

13 opengrok-indexer/src/main/java/org/opengrok/indexer/history/Repository.java

```
@@ -531,10 +531,7 @@ protected String getRepoRelativePath(final File file)

        @Override
        public StringBuffer format(Date date, StringBuffer toAppendTo, FieldPosition fieldPosition) {
-           for (DateFormat formatter : formatters) {
-               return formatter.format(date, toAppendTo, fieldPosition);
-           }
-           return toAppendTo.append("(date null)");
+           throw new UnsupportedOperationException("not implemented");
        }

        @Override
@@ -568,13 +565,7 @@ public Date parse(String source) throws ParseException {

        @Override
        public Date parse(String source, ParsePosition pos) {
-           Date d = null;
-           for (DateFormat formatter : formatters) {
-               if ((d = formatter.parse(source, pos)) != null) {
-                   return d;
-               }
-           }
-           return d;
+           throw new UnsupportedOperationException("not implemented");
        }
    }
}
```

Figure 10.2: Solution description and suggested code changes for bug report discussion from `https://github.com/oracle/opengrok/issues/2278`.

10.2. Suggesting Code Changes Based on Developer Discussions

The generated solution descriptions are intended to facilitate bug resolution by providing a high-level overview of the required changes. We are interested in taking the next step of providing further guidance through suggested code changes. For instance, currently, we focus on generating a NL description like "throw unsupported operation exception from unused repository date format methods" to help developers in better interpreting content from the bug report in Figure 10.2 that is relevant towards implementing the solution. Now, to help developers reason about how such a high-level idea should materialize as concrete code changes, we aim to generate suggested code changes which transform the `format`() and `parse`() methods into *dummy* methods by replacing the current body with a single statement: `throw new UnsupportedOperationException("not implemented")` (shown at right).

10.2.1. Problem Setting

To generate bug-fixing code changes, we must first link a given bug report discussion to the relevant, *buggy* code fragments within the code base. This can be achieved by either prompting a developer to manually locate this code or leveraging automatic bug localization systems (Saha et al., 2013; Rahman and Roy, 2018; Loyola et al., 2018; Zhu et al., 2020). Once we identify the buggy code, we could attempt to generate code changes by conditioning only this code, following prior work in bug fixing and applying common code change patterns (Section 2.3). However, such approaches blindly generate code changes, without reasoning about the broader context or capturing developer intent. Chakraborty and Ray (2021) recently found that incorporating code context from where the code fragment is extracted and also providing a natural language description of intent can lead to substantial improvements for bug fixing.

Inspired by this, we hypothesize that content within the bug report discussion can provide valuable context for generating bug-fixing code changes. Utterances in the discussion often contain code snippets, stack traces, and error messages (Li et al., 2018b) which serve as additional context for identifying specific code elements that are responsible for the bug that need to be edited. Additionally, the discussion can also shed light into intent, as we have shown that we can generate a natural language description of the solution using the same context. This description effectively captures the intent of the code changes. In fact, Chakraborty and Ray (2021) use commit messages as a proxy for natural language descriptions of intent, which is also a source of supervision for our task of generating solution descriptions (Section 7.2).

More concretely, for a given source code fragment S, we define S_b as the buggy version and S_f as the fixed version, after the bug is resolved. In Section 7.1, we defined the following task: $(\textit{title}, U_1, U_2, \dots) \rightarrow \textit{description}$. We now propose a new task for leveraging the discussion context and buggy code fragment to generate the fixed code, after applying the necessary code changes: $(\textit{title}, U_1, U_2, \dots, S_b) \rightarrow S_f$.

Additionally, since having a brief natural language description of intent is beneficial for learning code edits (Chakraborty and Ray, 2021; Tufano et al., 2021; Elgohary et al., 2021), we are interested in evaluating the value of incorporating such a concise description as another input: $(\textit{title}, U_1, U_2, \dots, S_b, \textit{description}) \rightarrow S_f$. For this, we plan to feed in the output of the best generation model from Chapter 7. To further study the extrinsic value of the generated description in capturing the solution, we intend to conduct the following experiment: $(S_b\ \textit{description}) \rightarrow S_f$. Chakraborty and Ray (2021)'s approach assumes that a human developer provides a natural language description specifying intent for making code changes. We will evaluate how our generated descriptions fair for the end task, relative to high-quality human descriptions as well as low-quality ones (e.g., generic or uninformative descriptions from Section 7.2.2).

Finally, we plan to integrate this task into a real-time setting, much like we do for the task of generating solution descriptions (Chapters 8 and 9).

10.2.2. Approaches

Chakraborty and Ray (2021) have shown that fine-tuning PLBART with additional context for generating bug fixes yields substantially higher performance than encoder-decoder models trained from scratch as well as other pretrained models. Since we find PLBART to also be effective in reasoning about the context within bug report discussions (Section 7.5), we believe that it serves as a good starting point for the proposed task.

However, this model failes to consider the *edit* nature of bug fixing in which much of S_b is preserved in S_f (Ding et al., 2020). As we discussed in Section 5.2.2, this unnecessarily burdens the decoder with generating unchanged tokens, which also increases the possibility of error propagation. To reduce this burden, we plan to first add explicit copying to PLBART, similar to how Einolghozati et al. (2020) incorporated a copy mechanism into BART for the task of learning to rephrase in dialogue systems. We believe this will better guide the model in learning to copy over tokens from the input. For this, we will explore token-based copying (Vinyals et al., 2015) as well as span-based copying (Panthaplackel et al., 2021a), which was shown to be useful for bug fixing.

We will further attempt to *eliminate* this burden by studying edit-based frameworks which are designed to only generate the edited tokens. Following the framework we developed in Section 5.2 for learning to edit comments, we will investigate the performance of fine-tuning PLBART to encode both the input context and decode a condensed edit sequence, S_e, which can then be aligned with S_b in order to produce S_f. Because S_e does not include unchanged tokens and includes many special tokens for specifying the edit type (e.g., `ReplaceOld`, `ReplaceNew`), the decoder's target output diverges from the continuous, coherent code sequences that are used for PLBART's pretraining. To account for this difference, we will likely need to first fine-tune PLBART on its original pretraining objectives using a large number of sequences structured like S_e, prior to fine-tuning on the bug fixing task.

However, by representing S_b and S_f as flattened sequences of code tokens, we discard rich structural context provided by the ASTs corresponding to these two code fragments. To inject structural information, we can represent them as flattened sequences of AST nodes, following the structure-based traversal (SBT) method proposed by Hu et al. (2018). We can produce an SBT representation of S_e by computing the condensed sequence of edits needed to transform S_b's SBT representation to S_f's SBT representation. Because PLBART is not pretrained on such representations, we will again likely need to first fine-tune it on the original pretraining objectives with a large number of SBT sequences.

Nonetheless, PLBART is not designed to reason about structured input or generate structured output. So, we plan to also explore more structured edit-based models that have been previously studied for various code editing tasks (Tarlow et al., 2020; Yao et al., 2021; Mesbah et al., 2019), which entail encoding the input context with graph-based models (Li et al., 2016) and decoding a series of AST edits.

10.2.3. Data and Evaluation

While we can extract code changes from commits associated with the bug reports that we collected (Section 7.2), we intend to first study this task in a more constrained setting. Tufano et al. (2019b) developed a dataset for studying this task, entailing simpler changes in small ($<$ 50 tokens) and medium (50-100 tokens) Java methods. There are 58,350 *small* examples (from 45,958 commits) and 65,455 *medium* examples (from 54,784 commits). We plan to focus on the examples in the dataset with commits which are linked to bug reports. Only 437 of the examples in Tufano et al. (2019b) have commits which overlap with the 141,334 commits in all of the bug reports we mined for our corpus. While this is a small number, there is very little overlap in the data we mined and that of Tufano et al. (2019b). Of the 58,454 projects they mined, there are only 263 which overlap with the 770 projects in our corpus. We intend to mine bug reports from the remaining projects in their corpus to obtain more examples. We also plan to apply their heuristics for extracting examples from commits to the commits in our corpus. Tufano et al. (2019b) released 10,054,468 bug-fixing commits, most of which are not used in their small and medium datasets. We plan to use bug-fix pairs from these unused commits for fine-tuning PLBART on the various representations mentioned in Section 10.2.2 with the original pretraining objectives.

Note that an "example" in the Tufano et al. (2019b) corpus does not signify a full bug fix. It is simply one set of code changes, among possibly many others, that are required for resolving the bug. Although generating one set is only a partial solution, we believe that it can still provide a starting point to developers in implementing the full solution. For evaluation, we will use exact match, and for cases in which all sets of code changes associated with a given commit are present in the corpus, we intend to also evaluate which fraction of them can be correctly generated.

Chapter 11

Conclusion

Software is constantly evolving to accommodate ever-changing technological user needs, wants, and concerns. To prevent software quality from deteriorating under the large volume of changes and also foster timely implementation of important changes, we aim to guide developers in making methodical software changes through natural language.

Inconsistent comments often materialize as a result of developers failing to update comments when they make changes to the corresponding body of code. To prevent such inconsistencies from forming, we first designed a deep learning approach for just-in-time inconsistency detection that encodes the syntactic structures of comments and code, which we showed to outperform various baselines as well as post hoc models that do not consider code changes. Next, we formulated the novel task of automatically updating inconsistent comments based on code changes, which we addressed through a framework that generates a sequence of edit actions by correlating cross-modal edits. We found that our approach outperforms multiple rule-based baselines and comment generation models, with respect to several automatic metrics and human evaluation. We further studied multiple techniques for combining the two tasks to build a comprehensive comment maintenance system that can detect and update inconsistent comments. For both tasks and the combined system, we observed that incorporating a set of salient features for explicitly associating comments and code substantially improves performance.

Next, when a software bug is reported, a discussion forms between developers to collaboratively resolve it. While the solution is often recommended within the discussion, this can get buried under a large amount of text. To enable developers to more easily locate and comprehend information relevant towards implementing the bug-resolving code changes and consequently expedite bug resolution, we presented our vision for an automated system which generates a concise description of the solution as soon as the necessary context becomes available in an ongoing developer discussion. Using a large dataset that we collected through supervision derived from commits and pull requests, we benchmarked approaches for generating informative solution descriptions. We also conducted a preliminary study on integrating such a generation model into a real-time setting by pipelining it with a classifier for determining when sufficient context emerges in an ongoing discussion. Through automated and human evaluation, we demonstrated the utility of these models and also highlighted their shortcomings, which we hope to address in future work.

Namely, as immediate next steps, we plan to develop an improved classification approach as well as a more intricate combined system which is jointly trained on generation and classification to allow the two interdependent tasks to better complement one another. This system learns to chime into a discussion at only one point in time and is not currently equipped to react to new activity after it performs generation. So, as our first long-term goal, we propose building an agent which *interactively* generates natural language descriptions to drive code changes. Finally, our second long-term goal is supplementing the high-level natural language description of the solution with actual suggestions for concrete code changes.

Bibliography

Ibrahim Abdelaziz, Julian Dolby, James P McCusker, and Kavitha Srinivas. 2020. Graph4code: A machine interpretable knowledge graph for code. *arXiv preprint arXiv:2002.09440*.

Rajas Agashe, Srinivasan Iyer, and Luke Zettlemoyer. 2019. JuICe: A large scale distantly supervised dataset for open domain context-based code generation. In *EMNLP-IJCNLP*, pages 5436–5446.

Karan Aggarwal, Tanner Rutgers, Finbarr Timbers, Abram Hindle, Russ Greiner, and Eleni Stroulia. 2015. Detecting duplicate bug reports with software engineering domain knowledge. In *SANER*, pages 211–220.

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. In *ACL*, pages 4998–5007.

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. In *NAACL-HLT*, pages 2655–2668.

Miltiadis Allamanis. 2019. The adverse effects of code duplication in machine learning models of code. In *SPLASH*, Onward!, pages 143–153.

Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018a. A survey of machine learning for big code and naturalness. *CSUR*, 51(4):1–37.

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018b. Learning to represent programs with graphs. In *ICLR*.

Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *ICML*, pages 2091–2100.

Miltiadis Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. 2015. Bimodal modelling of source code and natural language. In *ICML*, pages 2123–2132.

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating sequences from structured representations of code. In *ICLR*.

Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. 2020. Structural language models for any-code generation. In *ICML*.

Fernando Alva-Manchego, Louis Martin, Carolina Scarton, and Lucia Specia. 2019. EASSE: Easier automatic sentence simplification evaluation. In *EMNLP-IJCNLP: System Demonstrations*, pages 49–54.

John Anvik. 2006. Automating bug report assignment. In *ICSE*, pages 937–940.

Jude Arokiam and Jeremy S. Bradbury. 2020. Automatically predicting bug severity early in the development process. In *ICSE: New Ideas and Emerging Results*, pages 17–20.

Deeksha Arya, Wenting Wang, Jin L. C. Guo, and Jinghui Cheng. 2019. Analysis and detection of information types of open source software issue discussions. In *ICSE*, pages 454–464.

Abhijeet Awasthi, Sunita Sarawagi, Rasna Goyal, Sabyasachi Ghosh, and Vihari Piratla. 2019. Parallel iterative edit models for local sequence transduction. In *EMNLP-IJCNLP*, pages 4251–4261.

Muhammad Zubair Baloch, Shahid Hussain, Humaira Afzal, Muhammad Rafiq Mufti, and Bashir Ahmad. 2021. Software developer recommendation in terms of reducing bug tossing length. In *SpaCCS*, pages 396–407.

Satanjeev Banerjee and Alon Lavie. 2005. Meteor: An automatic metric for MT evaluation with improved correlation with human judgments. In *ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*, pages 65–72.

Olga Baysal, Michael W. Godfrey, and Robin Cohen. 2009. A bug you like: A framework for automated assignment of bugs. In *ICPC*, pages 297–298.

Taylor Berg-Kirkpatrick, David Burkett, and Dan Klein. 2012. An empirical investigation of statistical significance in NLP. In *EMNLP-CoNLL*, pages 995–1005.

Nick C. Bradley, Thomas Fritz, and Reid Holmes. 2018. Context-aware conversational developer assistants. In *ICSE*, pages 993–1003.

Shaked Brody, Uri Alon, and Eran Yahav. 2020. A structural model for contextual code changes. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–28.

Chris Brown and Chris Parnin. 2020. Understanding the impact of GitHub suggested changes on recommendations between developers. In *ESEC/FSE*, pages 1065–1076.

Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*.

Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2021. InferCode: Self-supervised learning of code representations by predicting subtrees. In *ICSE*, pages 1186–1197.

Luca Buratti, Saurabh Pujar, Mihaela Bornea, Scott McCarley, Yunhui Zheng, Gaetano Rossiello, Alessandro Morari, Jim Laredo, Veronika Thost, Yufan Zhuang, et al. 2020. Exploring software naturalness through neural language models. *arXiv preprint arXiv:2006.12641*.

Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When deep learning met code search. In *ESEC/FSE*, pages 964–974.

Joshua Charles Campbell, Abram Hindle, and José Nelson Amaral. 2014. Syntax errors just aren't natural: Improving error reporting with language models. In *MSR*, pages 252–261.

Saikat Chakraborty, Yangruibo Ding, Miltiadis Allamanis, and Baishakhi Ray. 2020. CODIT: Code editing with tree-based neural models. *TSE*.

Saikat Chakraborty and Baishakhi Ray. 2021. On multi-modal learning of editing source code. In *ASE*.

Krishna Kumar Chaturvedi and VB Singh. 2012. Determining bug severity using machine learning techniques. In *CONSEG*, pages 1–6.

Shobhit Chaurasia and Raymond Mooney. 2017. Dialog for language to code. In *IJCNLP*, pages 175–180.

Long Chen, Wei Ye, and Shikun Zhang. 2019. Capturing source code semantics via tree-based convolution over API-enhanced AST. In *International Conference on Computing Frontiers*, pages 174–182.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Ruey-Cheng Chen, Evi Yulianti, Mark Sanderson, and W. Bruce Croft. 2017. On the benefit of incorporating external features in a neural architecture for answer sentence selection. In *SIGIR*, pages 1017–1020.

Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *EMNLP*, pages 1724–1734.

Alfonso Cimasa, Anna Corazza, Carmen Coviello, and Giuseppe Scanniello. 2019. Word embeddings for comment coherence. In *SEAA*, pages 244–251.

Colin Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. 2020. PyMT5: multi-mode translation of natural language and python code with transformers. In *EMNLP*, pages 9052–9065.

Anna Corazza, Valerio Maggio, and Giuseppe Scanniello. 2018. Coherence of comments and method implementations: A dataset and an empirical investigation. *Software Quality Journal*, 26(2):751–777.

Chris Cummins, Zacharias Fisches, Tal Ben-Nun, Torsten Hoefler, Michael O'Boyle, and Hugh Leather. 2021. ProGraML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations. In *ICML*.

Milan Cvitkovic, Badal Singh, and Animashree Anandkumar. 2019. Open vocabulary learning on source code with a graph-structured cache. In *ICML*, pages 1475–1485.

Samip Dahal, Adyasha Maharana, and Mohit Bansal. 2021. Analysis of tree-structured architectures for code generation. In *Findings of ACL-IJCNLP*, pages 4382–4391.

Giuseppe Destefanis, Marco Ortu, David Bowes, Michele Marchesi, and Roberto Tonelli. 2018. On measuring affects of GitHub issues' commenters. In *International Workshop on Emotion Awareness in Software Engineering*, pages 14–19.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT*, pages 4171–4186.

Jin Ding, Hailong Sun, Xu Wang, and Xudong Liu. 2018. Entity-level sentiment analysis of issue comments. In *International Workshop on Emotion Awareness in Software Engineering*, pages 7–13.

Yangruibo Ding, Baishakhi Ray, Premkumar Devanbu, and Vincent J. Hellendoorn. 2020. Patching as translation: The data and the metaphor. In *ASE*, pages 275–286.

Li Dong and Mirella Lapata. 2016. Language to logical form with neural attention. In *ACL*, pages 33–43.

Yue Dong, Zichao Li, Mehdi Rezagholizadeh, and Jackie Chi Kit Cheung. 2019. EditNTS: An neural programmer-interpreter model for sentence simplification through explicit editing. In *ACL*, pages 3393–3402.

Arash Einolghozati, Anchit Gupta, Keith Diedrick, and Sonal Gupta. 2020. Sound natural: Content rephrasing in dialog systems. In *EMNLP*, pages 5101–5108.

Ahmed Elgohary, Christopher Meek, Matthew Richardson, Adam Fourney, Gonzalo Ramos, and Ahmed Hassan Awadallah. 2021. NL-EDIT: Correcting semantic parse errors through natural language interaction. In *NAACL-HLT*, pages 5599–5610.

Ayesha Enayet and Gita Sukthankar. 2020. A transfer learning approach for dialogue act classification of GitHub issue comments. *arXiv preprint arXiv:2011.04867*.

Khashayar Etemadi and Martin Monperrus. 2020. On the relevance of cross-project learning with nearest neighbours for commit message generation. In *ICSE Workshops*, pages 470–475.

Yuanrui Fan, Xin Xia, David Lo, and Ahmed E. Hassan. 2020. Chaff from the wheat: Characterizing and determining valid bug reports. *TSE*, 46(5):495–525.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of EMNLP*, pages 1536–1547.

Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Structured neural summarization. In *ICLR*.

Beat Fluri, Michael Wursch, and Harald C Gall. 2007. Do code and comments co-evolve? On the relation between source code and comment changes. In *WCRE*, pages 70–79.

Beat Fluri, Michael Würsch, Emanuel Giger, and Harald C. Gall. 2009. Analyzing the co-evolution of comments and source code. *Software Quality Journal*, 17(4):367–394.

Stephen R. Foster, William G. Griswold, and Sorin Lerner. 2012. WitchDoctor: IDE support for real-time auto-completion of refactorings. In *ICSE*, pages 222–232.

Xi Ge, Quinton L. DuBose, and Emerson Murphy-Hill. 2012. Reconciling manual and automatic refactoring. In *ICSE*, pages 211–221.

R. Stuart Geiger, Kevin Yu, Yanlai Yang, Mindy Dai, Jie Qiu, Rebekah Tang, and Jenny Huang. 2020. Garbage in, garbage out? Do machine learning application papers in social computing report where human-labeled training data comes from? In *Conference on Fairness, Accountability, and Transparency*, pages 325–336.

Mehdi Golzadeh, Alexandre Decan, and Tom Mens. 2019. On the effect of discussions on pull request decisions. In *Belgium-Netherlands Software Evolution Workshop*.

Luiz Alberto Ferreira Gomes, Ricardo da Silva Torres, and Mario Lúcio Côrtes. 2019. Bug report severity level prediction in open source software: A survey and research opportunities. *Information and Software Technology*, 115:58–78.

Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *ICSE*, pages 933–944.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Jian Yin, Daxin Jiang, and M. Zhou. 2021. GraphCodeBERT: Pre-training code representations with data flow. In *ICLR*.

Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing common c language errors by deep learning. In *AAAI*.

Izzeddin Gur, Semih Yavuz, Yu Su, and Xifeng Yan. 2018. DialSQL: Dialogue based structured query generation. In *ACL*, pages 1339–1349.

Rajarshi Haldar, Lingfei Wu, JinJun Xiong, and Julia Hockenmaier. 2020. A multi-perspective architecture for semantic code search. In *ACL*, pages 8563–8568.

Jianjun He, Ling Xu, Yuanrui Fan, Zhou Xu, Meng Yan, and Yan Lei. 2020. Deep learning based valid bug reports determination and explanation. In *ISSRE*, pages 184–194.

Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. 2020. Global relational models of source code. In *ICLR*.

Vincent J. Hellendoorn, Jason Tsay, Manisha Mukherjee, and Martin Hirzel. 2021. Towards automating code review at scale. In *ESEC/FSE*, pages 1479–1482.

Abram Hindle and Curtis Onuczko. 2019. Preventing duplicate bug reports by continuously querying bug reports. *Empirical Software Engineering*, 24(2):902–936.

Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. 2020. Cc2vec: Distributed representations of code changes. In *ICSE*, pages 518–529.

Ari Holtzman, Jan Buys, Maxwell Forbes, and Yejin Choi. 2020. The curious case of neural text degeneration. In *ICLR*.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *ICPC*, pages 200–210. IEEE.

LiGuo Huang, Vincent Ng, Isaac Persing, Ruili Geng, Xu Bai, and Jeff Tian. 2011. Autoodc: Automated generation of orthogonal defect classifications. In *ASE*, pages 412–415.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Walid M. Ibrahim, Nicolas Bettenburg, Bram Adams, and Ahmed E. Hassan. 2012. On the relationship between comment update practices and software bugs. *Journal of Systems and Software*, 85(10):2293–2304.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *ACL*, pages 2073–2083.

Oskar Jarczyk, Blazej Gruszka, Szymon Jaroszewicz, Leszek Bukowski, and Adam Wierzbicki. 2014. Github projects. Quality analysis of open-source software. In *SocInfo*, pages 80–94.

He Jiang, Jingxuan Zhang, Hongjing Ma, Najam Nazar, and Zhilei Ren. 2017. Mining authorship characteristics in bug repositories. *Science China Information Sciences*, 60(1):1–16.

Jing Jiang, Jiateng Zheng, Yun Yang, Li Zhang, and Jie Luo. 2021a. Predicting accepted pull requests in GitHub. *Science China Information Sciences*, 64.

Xue Jiang, Zhuoran Zheng, Chen lv, Liang Li, and Lei Lyu. 2021b. TreeBERT: A tree-based pre-trained model for programming language. In *UAI*.

Zhen Ming Jiang and Ahmed E. Hassan. 2006. Examining the evolution of code comments in PostgreSQL. In *MSR*, pages 179–180.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *ICML*.

Rafael-Michael Karampatsis and Charles Sutton. 2020a. How often do single-statement bugs occur? The ManySStuBs4J dataset. In *MSR*, pages 573–577.

Rafael-Michael Karampatsis and Charles Sutton. 2020b. Scelmo: Source code embeddings from language models. *arXiv preprint arXiv:2004.13214*.

David Kavaler, Sasha Sirovica, Vincent Hellendoorn, Raul Aranovich, and Vladimir Filkov. 2017. Perceived language complexity in GitHub issue discussions and their effect on issue resolution. In *ASE*, pages 72–83.

Yalin Ke, Kathryn T Stolee, Claire Le Goues, and Yuriy Brun. 2015. Repairing programs with semantic code search. In *ASE*, pages 295–306.

Ninus Khamis, René Witte, and Juergen Rilling. 2010. Automatic quality assessment of source code comments: The JavadocMiner. In *International Conference on Application of Natural Language to Information Systems*, pages 68–79.

Riivo Kikas, Marlon Dumas, and Dietmar Pfahl. 2015. Issue dynamics in GitHub projects. In *PROFES*, pages 295–310.

Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *ICSE*, pages 802–811.

Wei-Jen Ko, Greg Durrett, and Junyi Jessy Li. 2019. Linguistically-informed specificity and semantic plausibility for dialogue generation. In *NAACL-HLT*, pages 3456–3466.

Oleksii Kononenko, Tresa Rose, Olga Baysal, Michael Godfrey, Dennis Theisen, and Bart de Water. 2018. Studying pull request merges: A case study of shopify's active merchant. In *ICSE: Software Engineering in Practice Track*, pages 124–133.

Klaus Krippendorff. 2011. Computing Krippendorff's alpha reliability. Technical report, University of Pennsylvania.

Rajiv Krishnamurthy, Varghese Jacob, Suresh Radhakrishnan, and Kutsal Dogan. 2016. Peripheral developer participation in open source projects: an empirical analysis. *TMIS*, 6(4):1–31.

Reno Kriz, João Sedoc, Marianna Apidianaki, Carolina Zheng, Gaurav Kumar, Eleni Miltsakaki, and Chris Callison-Burch. 2019. Complexity-weighted loss and diverse reranking for sentence simplification. In *NAACL-HLT*, pages 3137–3147.

Ahmed Lamkanfi, Serge Demeyer, Emanuel Giger, and Bart Goethals. 2010. Predicting the severity of a reported bug. In *MSR*, pages 1–10.

Guillaume Lample, Miguel Ballesteros, Sandeep Subramanian, Kazuya Kawakami, and Chris Dyer. 2016. Neural architectures for named entity recognition. In *NAACL-HLT*, pages 260–270.

Alina Lazar, Sarah Ritchey, and Bonita Sharif. 2014. Improving the accuracy of duplicate bug report detection using textual similarity measures. In *MSR*, pages 308–311.

Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: Syntax- and semantic-guided repair synthesis via programming by examples. In *FSE*, pages 593–604.

Xuan Bach D. Le, David Lo, and Claire Le Goues. 2016. History driven program repair. In *SANER*, volume 1, pages 213–224.

Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A generic method for automatic software repair. *TSE*, 38(1):54–72.

Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. In *ICPC*, pages 184–195.

Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *ICSE*, pages 795–806.

Meir Lehman and Juan Fernáandez-Ramil. 2006. *Software Evolution*, pages 7–40. John Wiley & Sons.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *ACL*, pages 7871–7880.

Juncen Li, Robin Jia, He He, and Percy Liang. 2018a. Delete, retrieve, generate: a simple approach to sentiment and style transfer. In *NAACL-HLT*, pages 1865–1874.

Junyi Jessy Li and Ani Nenkova. 2015. Fast and accurate prediction of sentence specificity. In *AAAI*, pages 2281–2287.

Xiaochen Li, He Jiang, Dong Liu, Zhilei Ren, and Ge Li. 2018b. Unsupervised deep bug report summarization. In *ICPC*, pages 144–155.

Xiaolong Li and Kristy Boyer. 2015. Semantic grounding in dialogue for complex problem solving. In *NAACL-HLT*, pages 841–850.

Xiaolong Li and Kristy Boyer. 2016. Reference resolution in situated dialogue with learned semantics. In *SIGDIAL*, pages 329–338.

Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. 2016. Gated graph sequence neural networks. In *ICLR*.

Zhixing Li, Yue Yu, Gang Yin, T. Wang, Qiang Fan, and Huaimin Wang. 2017. Automatic classification of review comments in pull-based development model. In *SEKE*, pages 572–577.

Yuding Liang and Kenny Zhu. 2018. Automatic generation of text descriptive comments for code blocks. In *AAAI*, volume 32.

Chin-Yew Lin. 2004. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81.

Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D Ernst. 2018. NL2Bash: A corpus and semantic parser for natural language interface to the linux operating system. In *LREC*.

Chia-Wei Liu, Ryan Lowe, Iulian Serban, Mike Noseworthy, Laurent Charlin, and Joelle Pineau. 2016. How NOT to evaluate your dialogue system: An empirical study of unsupervised evaluation metrics for dialogue response generation. In *EMNLP*, pages 2122–2132.

Haoran Liu, Yue Yu, Shanshan Li, Yong Guo, Deze Wang, and Xiaoguang Mao. 2020. BugSum: Deep context understanding for bug report summarization. In *ICPC*, pages 94–105.

Xiaoyu Liu, LiGuo Huang, Chuanyi Li, and Vincent Ng. 2018a. Linking source code to untangled change intents. In *ICSME*, pages 393–403.

Zhiyong Liu, Huanchao Chen, Xiangping Chen, Xiaonan Luo, and Fan Zhou. 2018b. Automatic detection of outdated comments during code changes. In *COMPSAC*, pages 154–163.

Zhongxin Liu, Xin Xia, Christoph Treude, David Lo, and Shanping Li. 2019. Automatic generation of pull request descriptions. In *ASE*, pages 176–188.

Ryan Lowe, Nissan Pow, Iulian Serban, and Joelle Pineau. 2015. The Ubuntu dialogue corpus: A large dataset for research in unstructured multi-turn dialogue systems. In *SIGDIAL*, pages 285–294.

Pablo Loyola, Kugamoorthy Gajananan, and Fumiko Satoh. 2018. Bug localization by learning to rank and represent bug inducing changes. In *CIKM*, pages 657–665.

Pablo Loyola, Edison Marrese-Taylor, and Yutaka Matsuo. 2017. A neural architecture for generating natural language descriptions from source code changes. In *ACL*, pages 287–292.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective approaches to attention-based neural machine translation. In *EMNLP*, pages 1412–1421.

Chandra Maddila, Sai Surya Upadrasta, Chetan Bansal, Nachiappan Nagappan, Georgios Gousios, and Arie van Deursen. 2020. Nudge: Accelerating overdue pull requests towards completion. *arXiv preprint arXiv:2011.12468*.

Haroon Malik, Istehad Chowdhury, Hsiao-Ming Tsou, Zhen Ming Jiang, and Ahmed E. Hassan. 2008. Understanding the rationale for updating a function's comment. In *ICSME*, pages 167–176.

Nikita Mehrotra, Navdha Agarwal, Piyush Gupta, Saket Anand, David Lo, and Rahul Purandare. 2021. Modeling functional similarity in source code with graph-based siamese networks. *TSE*.

Na Meng, Lisa Hua, Miryung Kim, and Kathryn S. McKinley. 2015. Does automated refactoring obviate systematic editing? In *ICSE*, pages 392–402.

Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. 2019. DeepDelta: Learning to repair compilation errors. In *ESEC/FSE*, pages 925–936.

Anders Miltner, Sumit Gulwani, Vu Le, Alan Leung, Arjun Radhakrishna, Gustavo Soares, Ashish Tiwari, and Abhishek Udupa. 2019. On the fly synthesis of edit suggestions. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29.

Ruslan Mitkov. 1999. Anaphora resolution: The state of the art. Technical report.

Dana Movshovitz-Attias and William Cohen. 2013. Natural language models for predicting programming comments. In *ACL*, pages 35–40.

Emerson Murphy-Hill, Thomas Zimmermann, Christian Bird, and Nachiappan Nagappan. 2015. The design space of bug fixes and how developers navigate it. *TSE*, 41(1):65–81.

Aravind Nair, Avijit Roy, and Karl Meinke. 2020. funcGNN: A graph neural network approach to program similarity. In *ESEM*, pages 1–11.

Ramesh Nallapati, Feifei Zhai, and Bowen Zhou. 2017. Summarunner: A recurrent neural network based sequence model for extractive summarization of documents. In *AAAI*, pages 3075–3081.

Courtney Napoles, Keisuke Sakaguchi, Matt Post, and Joel Tetreault. 2015. Ground truth for grammatical error correction metrics. In *ACL-IJCNLP*, pages 588–593.

Graham Neubig, Makoto Morishita, and Satoshi Nakamura. 2015. Neural reranking improves subjective quality of machine translation: NAIST at WAT2015. In *Workshop on Asian Translation*, pages 35–41.

Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N. Nguyen, and Danny Dig. 2016. API code recommendation using statistical learning from fine-grained changes. In *FSE*, pages 511–522.

Anh Tuan Nguyen and Tien N. Nguyen. 2015. Graph-based statistical language model for code. In *ICSE*, pages 858–868.

Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2012. Multi-layered approach for recovering links between bug reports and fixes. In *FSE*, pages 63:1–63:11.

Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson Jr, Anh Tuan Nguyen, Miryung Kim, and Tien N Nguyen. 2010. A graph-based approach to api usage adaptation. *SIGPLAN Notices*, 45(10):302–321.

Pengyu Nie, Rishabh Rai, Junyi Jessy Li, Sarfraz Khurshid, Raymond J. Mooney, and Milos Gligoric. 2019. A framework for writing trigger-action todo comments in executable format. In *ESEC/FSE*, pages 385–396.

Pengyu Nie, Jiyang Zhang, Junyi Jessy Li, Raymond J Mooney, and Milos Gligoric. 2021. Evaluation methodologies for code learning tasks. *arXiv preprint arXiv:2108.09619*.

Yuki Noyori, Hironori Washizaki, Yoshiaki Fukazawa, Keishi Ooshima, Hideyuki Kanuka, Shuhei Nojiri, and Ryosuke Tsuchiya. 2019. What are good discussions within bug report comments for shortening bug fixing time? In *QRS*, pages 280–287.

Ally S. Nyamawe, Hui Liu, Nan Niu, Qasim Umer, and Zhendong Niu. 2019. Automated recommendation of software refactorings based on feature requests. In *International Requirements Engineering Conference*, pages 187–198.

Paul Oman and Jack Hagemeister. 1992. Metrics for assessing a software system's maintainability. In *Conference on Software Maintenance*, pages 337–338.

Oracle. 2020. Javadoc. `https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html`.

Oracle. 2021. Comments. `https://www.oracle.com/java/technologies/javase/codeconventions-comments.html`.

Sebastiano Panichella, Gerardo Canfora, and Andrea Di Sorbo. 2021. "Won't we fix this issue?" Qualitative characterization and automated identification of wontfix issues on github. *Information and Software Technology*, 139:106665.

Sheena Panthaplackel, Miltiadis Allamanis, and Marc Brockschmidt. 2021a. Copy that! editing sequences by copying spans. In *AAAI*, pages 13622–13630.

Sheena Panthaplackel, Milos Gligoric, Raymond J. Mooney, and Junyi Jessy Li. 2020a. Associating natural language comment and source code entities. In *AAAI*, pages 8592–8599.

Sheena Panthaplackel, Junyi Jessy Li, Milos Gligoric, and Raymond J. Mooney. 2021b. Deep just-in-time inconsistency detection between comments and source code. In *AAAI*, pages 427–435.

Sheena Panthaplackel, Junyi Jessy Li, Milos Gligoric, and Raymond J. Mooney. 2021c. Learning to describe solutions for bug reports based on developer discussions. *arXiv preprint arXiv:2110.04353*.

Sheena Panthaplackel, Pengyu Nie, Milos Gligoric, Junyi Jessy Li, and Raymond Mooney. 2020b. Learning to update natural language comments based on code changes. In *ACL*, pages 1853–1868.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a method for automatic evaluation of machine translation. In *ACL*, pages 311–318.

Luca Pascarella and Alberto Bacchelli. 2017. Classifying code comments in java open-source software systems. In *MSR*, pages 227–237.

Rebecca Passonneau. 2006. Measuring agreement on set-valued items (MASI) for semantic and pragmatic annotation. In *LREC*.

Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. In *NAACL-HLT*, pages 2227–2237.

Fazle Rabbi and Md Saeed Siddik. 2020. Detecting code comment inconsistency using siamese recurrent network. In *ICPC - Early Research Achievements*, pages 371–375.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. In *ACL*, pages 1139–1149.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67.

Mohammad Masudur Rahman and Chanchal K Roy. 2018. Improving ir-based bug localization with context-aware query reformulation. In *ESEC/FSE*, pages 621–632.

Sarah Rastkar, Gail C. Murphy, and Gabriel Murray. 2014. Automatic summarization of bug reports. *TSE*, 40(4):366–380.

Inderjot Kaur Ratol and Martin P. Robillard. 2017. Detecting fragile comments. *ASE*, pages 112–122.

Veselin Raychev, Max Schäfer, Manu Sridharan, and Martin Vechev. 2013. Refactoring with synthesis. *SIGPLAN Notices*, 48(10):339–354.

Xiaoxue Ren, Zhenchang Xing, Xin Xia, David Lo, Xinyu Wang, and John Grundy. 2019. Neural network-based detection of self-admitted technical debt: From performance to explainability. *TOSEM*, 28:1–45.

Gema Rodríguez-Pérez, Gregorio, Robles, Alexander Serebrenik, Andy, Zaidman, Daniel M. Germán, Jesus, and Jesus M. Gonzalez-Barahona. 2020. How bugs are born: A model to identify how bugs are introduced in software components. *Empirical Software Engineering*, 25:1294–1340.

Ankita Sadu. 2019. Automatic detection of outdated comments in open source Java projects. Master's thesis, Universidad Politécnica de Madrid.

Ripon K Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E Perry. 2013. Improving bug localization using structured information retrieval. In *ASE*, pages 345–355. IEEE.

Abigail See, Peter J. Liu, and Christopher D. Manning. 2017. Get to the point: Summarization with pointer-generator networks. In *ACL*, pages 1073–1083.

Iulian V. Serban, Alessandro Sordoni, Yoshua Bengio, Aaron Courville, and Joelle Pineau. 2016. Building end-to-end dialogue systems using generative hierarchical neural network models. In *AAAI*, pages 3776–3783.

Eui Chul Shin, Miltiadis Allamanis, Marc Brockschmidt, and Alex Polozov. 2019. Program synthesis and semantic parsing with learned code idioms. *NeurIPS*, 32:10825–10835.

Richard Shin, Illia Polosukhin, and Dawn Song. 2018. Towards specification-directed program repair. In *ICLR Workshop*.

Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. 2010. Towards automatically generating summary comments for java methods. In *ASE*, pages 43–52.

Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. 2011. Generating parameter comments and integrating with method summaries. In *ICPC*, pages 71–80.

Nataliia Stulova, Arianna Blasi, Alessandra Gorla, and Oscar Nierstrasz. 2020. Towards detecting inconsistent comments in java source code automatically. In *SCAM*, pages 65–69.

Akhilesh Sudhakar, Bhargav Upadhyay, and Arjun Maheswaran. 2019. "Transforming" delete, retrieve, generate approach for controlled text style transfer. In *EMNLP-IJCNLP*, pages 3267–3277.

Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2020. TreeGen: A tree-based transformer architecture for code generation. In *AAAI*, pages 8984–8991.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *NeurIPS*, pages 3104–3112.

Adam Svensson. 2015. Reducing outdated and inconsistent code comments during software development: The comment validator program. Master's thesis, Uppsala University.

Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. IntelliCode compose: Code generation using transformer. In *ESEC/FSE*, pages 1433–1443.

Jeniya Tabassum, Mounica Maddela, Wei Xu, and Alan Ritter. 2020. Code and named entity recognition in StackOverflow. In *ACL*, pages 4913–4926.

Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /*iComment: Bugs or bad comments?*/. In *SOSP*, pages 145–158.

Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. 2011. aComment: Mining annotations from comments and code to detect interrupt related concurrency bugs. In *ICSE*, pages 11–20.

Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. 2012. @tComment: Testing javadoc comments to detect comment-code inconsistencies. In *ICST*, pages 260–269.

Xin Tan, Minghui Zhou, and Zeyu Sun. 2020. A first look at good first issues on github. In *ESEC/FSE*, pages 398–409.

Wesley Tansey and Eli Tilevich. 2008. Annotation refactoring: Inferring upgrade transformations for legacy applications. *SIGPLAN Notices*, 43(10):295–312.

Daniel Tarlow, Subhodeep Moitra, Andrew Rice, Zimin Chen, Pierre-Antoine Manzagol, Charles Sutton, and Edward Aftandilian. 2020. Learning to fix build errors with Graph2Diff neural networks. In *ICSE Workshops*, pages 19–20.

Ted Tenny. 1988. Program readability: Procedures versus comments. *TSE*, 14(9):1271–1279.

Ferdian Thung, David Lo, and Lingxiao Jiang. 2012. Automatic defect categorization. In *WCRE*, pages 205–214.

Yuan Tian, David Lo, and Chengnian Sun. 2012a. Information retrieval based nearest neighbor classification for fine-grained bug severity prediction. In *WCRE*, pages 215–224.

Yuan Tian, Chengnian Sun, and David Lo. 2012b. Improved duplicate bug report identification. In *CSMR*, pages 385–390.

Jason Tsay, Laura Dabbish, and James Herbsleb. 2014. Let's talk about it: Evaluating contributions through discussion in GitHub. In *FSE*, pages 144–154.

Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. 2019a. On learning meaningful code changes via neural machine translation. In *ICSE*, pages 25–36.

Michele Tufano, Cody Watson, G. Bavota, M. D. Penta, Martin White, and D. Poshyvanyk. 2019b. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *TOSEM*, 28:1–29.

Rosalia Tufano, Luca Pascarella, Michele Tufano, Denys Poshyvanyk, and Gabriele Bavota. 2021. Towards automating code review activities. In *ICSE*.

Daniel E. Turk, Robert B. France, and Bernhard Rumpe. 2005. Assumptions underlying agile software-development processes. *Journal of Database Management*, 16:62–87.

Erik van der Veen, Georgios Gousios, and Andy Zaidman. 2015. Automatically prioritizing pull requests. In *MSR*, pages 357–361.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *NeurIPS*, volume 30.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer networks. In *NeurIPS*, pages 2692–2700.

Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020a. RAT-SQL: Relation-aware schema encoding and linking for text-to-SQL parsers. In *ACL*, pages 7567–7578.

Wenhan Wang, Ge Li, Sijie Shen, Xin Xia, and Zhi Jin. 2020b. Modular tree network for source code representation learning. *TOSEM*, 29(4):1–23.

Wenhan Wang, Kechi Zhang, Ge Li, and Zhi Jin. 2020c. Learning to represent programs with heterogeneous graphs. *arXiv preprint arXiv:2012.04188*.

Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. 2020. LambdaNet: Probabilistic type inference using graph neural networks. In *ICLR*.

Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. 2019. A large-scale empirical study on code-comment inconsistencies. In *ICPC*, pages 53–64.

Mairieli Wessel, Alexander Serebrenik, Igor Wiese, Igor Steinmacher, and Marco A. Gerosa. 2020. Effects of adopting code review bots on pull requests to oss projects. In *ICSME*, pages 1–11.

Andrew Wood, Paige Rodeghero, Ameer Armaly, and Collin McMillan. 2018. Detecting speech act types in developer question/answer conversations during bug repair. In *ESEC/FSE*, pages 491–502.

Scott N Woodfield, Hubert E Dunsmore, and Vincent Yun Shen. 1981. The effect of modularization and comments on program comprehension. In *ICSE*, pages 215–223.

Shengqu Xi, Yuan Yao, Xusheng Xiao, Feng Xu, and Jian Lu. 2018. An effective approach for routing the bug reports to the right fixers. In *Asia-Pacific Symposium on Internetware*, pages 1–10.

Frank F Xu, Zhengbao Jiang, Pengcheng Yin, Bogdan Vasilescu, and Graham Neubig. 2020. Incorporating external knowledge through pre-training for natural language to code generation. In *ACL*, pages 6045–6052.

Shengbin Xu, Yuan Yao, Feng Xu, Tianxiao Gu, Hanghang Tong, and Jian Lu. 2019a. Commit message generation for source code changes. In *IJCAI*, pages 3975–3981.

Sihan Xu, Sen Zhang, Weijing Wang, Xinya Cao, Chenkai Guo, and Jing Xu. 2019b. Method name suggestion with hierarchical attention networks. In *SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 10–21.

Wei Xu, Courtney Napoles, Ellie Pavlick, Quanze Chen, and Chris Callison-Burch. 2016. Optimizing statistical machine translation for text simplification. *TACL*, 4:401–415.

Huong Nguyen Thi Xuan, Vo Cong Hieu, and Anh-Cuong Le. 2018. Adding external features to convolutional neural network for aspect-based sentiment analysis. In *NICS*, pages 53–59.

Cheng-Zen Yang, Kun-Yu Chen, Wei-Chen Kao, and Chih-Chuan Yang. 2014. Improving severity prediction on software bug reports using quality indicators. In *ICSESS*, pages 216–219.

Ziyu Yao, Yu Su, Huan Sun, and Wen-tau Yih. 2019. Model-based interactive semantic parsing: A unified framework and a text-to-SQL case study. In *EMNLP-IJCNLP*, pages 5447–5458.

Ziyu Yao, Daniel S Weld, Wei-Peng Chen, and Huan Sun. 2018. StaQC: A systematically mined question-code dataset from stack overflow. In *WWW*, pages 1693–1703.

Ziyu Yao, Frank F. Xu, Pengcheng Yin, Huan Sun, and Graham Neubig. 2021. Learning structural edits via incremental tree transformations. In *ICLR*.

Michihiro Yasunaga and Percy Liang. 2020. Graph-based, self-supervised program repair from diagnostic feedback. In *ICML*.

Xi Ye, Qiaochu Chen, Xinyu Wang, Isil Dillig, and Greg Durrett. 2020. Sketch-driven regular expression generation from natural language and examples. *TACL*, 8:679–694.

Yunwen Ye and Kouichi Kishida. 2003. Toward an understanding of the motivation open source software developers. In *ICSE*, pages 419–429.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from Stack Overflow. In *MSR*, pages 476–486.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. In *ACL*, pages 440–450.

Pengcheng Yin, Graham Neubig, Miltiadis Allamanis, Marc Brockschmidt, and Alexander L. Gaunt. 2019. Learning to represent edits. In *ICLR*.

Xiaohan Yu, Quzhe Huang, Zheng Wang, Yansong Feng, and Dongyan Zhao. 2020. Towards context-aware code comment generation. In *Findings of EMNLP*, pages 3938–3947.

Yue Yu, Huaimin Wang, Vladimir Filkov, Premkumar Devanbu, and Bogdan Vasilescu. 2015. Wait for it: Determinants of pull request evaluation latency on github. In *MSR*, pages 367–371.

Yue Yu, Huaimin Wang, Gang Yin, and Charles X. Ling. 2014. Reviewer recommender of pull-requests in GitHub. In *ICSME*, pages 609–612.

Feng Zhang, Foutse Khomh, Ying Zou, and Ahmed E. Hassan. 2012. An empirical study on factors impacting bug fixing time. In *WCRE*, pages 225–234.

Neng Zhang, Qiao Huang, Xin Xia, Ying Zou, David Lo, and Zhenchang Xing. 2020. Chatbot4QR: Interactive query refinement for technical question retrieval. *TSE*.

Ruqing Zhang, Jiafeng Guo, Yixing Fan, Yanyan Lan, Jun Xu, and Xueqi Cheng. 2018. Learning to control the specificity in neural response generation. In *ACL*, pages 1108–1117.

Guoliang Zhao, Daniel Alencar da Costa, and Ying Zou. 2019. Improving the pull requests review process using learning-to-rank algorithms. *Empirical Software Engineering*, 24:2140–2170.

Jie Zhao and Huan Sun. 2020. Adversarial training for code retrieval with question-description relevance regularization. In *Findings of EMNLP*, pages 4049–4059.

Yu Zhou, Gu Ruihang, Chen Taolue, Huang Zhiqiu, Panichella Sebastiano, and Gall Harald. 2017. Analyzing APIs documentation and code to detect directive defects. In *ICSE*, pages 27–37.

Ziye Zhu, Yun Li, Hanghang Tong, and Yu Wang. 2020. CooBa: Cross-project bug localization via adversarial transfer learning. In *IJCAI*, pages 3565–3571.