The Dissertation Committee for Sheena Liz Panthaplackel

certifies that this is the approved version of the following dissertation:

Facilitating Software Evolution through Natural Language Comments and Dialogue

Committee:

Raymond J. Mooney, Supervisor

Junyi Jessy Li, Co-Supervisor

Milos Gligoric

Greg Durrett

Charles Sutton

Facilitating Software Evolution through Natural Language Comments and Dialogue

by

Sheena Liz Panthaplackel

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2022

Acknowledgments

As these five years come to an end, I cannot but think of the many individuals whose support have made this thesis possible. First and foremost, I would like to sincerely thank my advisors Raymond J. Mooney and Junyi Jessy Li for their time and energy in building me as a researcher over the last five years. I would like to thank Ray for his broad vision that helped shape my research, for teaching me to not be afraid to pursue new tasks that have not been previously studied, and for allowing me immense freedom in structuring my research. I would like to thank Jessy for guiding me in designing technical approaches, for teaching me how to frame ideas when writing research papers, and for being patient with me through all my last minute questions and concerns. Additionally, I would like to thank Milos Gligoric for helping me identify useful and meaningful problems to solve, for working with me extensively to design user studies, and for all his feedback throughout these years. I would like to also thank Greg Durrett and Charles Sutton for serving on my thesis committee and for insightful discussions and feedback.

I would like to thank my mentors at Microsoft Research Cambridge, Miltos Allamanis and Marc Brockschmidt for all that they have taught me during the early stages of the PhD, which continue to influence me in the way I formulate research problems, approach technical challenges, and face disheartening outcomes. I would like to thank my mentors at Bloomberg, Adrian Benton and Mark Dredze, for getting me excited about new problems outside my area of focus and for guid-

ing me through a completely new terrain. I would also like to thank Bloomberg for generously granting me the Data Science Fellowship that has funded my research.

I would like to thank my lab mates Angela Lin, Prasoon Goyal, Jialin Wu, Aishwarya Padmakumar and collaborators Pengyu Nie and Jiyang Zhang for discussions and technical feedback that have greatly influenced my research. I would like to thank my grad school friends, Tanya Goyal, Soujanya Ponnapalli, Shailee Jain, Tushar Nagarajan, and Keya Ghonasgi, for keeping me sane and making this experience so memorable. I would like to thank my fellow interns at Microsoft Research Cambridge, Goutham Ramakrishnan, Swathi Jagannath, Rushil Khurana, Padmaja Jonnalagedda, Praneeth Chakravarthula, Arjun Kashyap, and Arpita Biswas, for all their support and encouragement during the early stages of the PhD. I would like to also thank the many individuals who have supported me and given me feedback from within the broader UT NLP community, Eunsol Choi, Yasumasa Onoe, Jiacheng Xu, Jifan Chen, Eric Holgate, Pengxiang Cheng, Shrey Desai, and Anubrata Das, as well as within the borader SE community, Nader Al Awar, Yu Yuki Liu, Zhiqiang Zhang, Kush Jain, Steven Zhu, and Marko Vasic. I would like to also thank my friends back home in Chicago, Varsha John, Jasmine Puthuvelil, Jenny James, Chris Mathew, Priscilla Alex, Lidiya Jacob, Jagan Jimmy, and Anna Chirayil, for their continuous support throughout the years.

Finally, I would not have reached this point without the incredible love and support that I received from my family. I would like to thank my aunts, uncles, and cousins for being the greatest cheerleaders. I would like to thank my little brother, Kevin, for being my biggest fan and for always brightening up my day. I would like

to thank my mom for her unwavering support and patience through these years, for comforting me and taking care of me in the most stressful times, and for instilling in me the value of hard work. I would like to thank my dad for being my teacher ever since I can remember, for encouraging me to be curious and ask questions, for pushing me to dream big, and for being the greatest inspiration of my life.

Facilitating Software Evolution through Natural Language Comments and Dialogue

Sheena Liz Panthaplackel, Ph.D.
The University of Texas at Austin, 2022

Supervisors: Raymond J. Mooney and Junyi Jessy Li

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 supporting and driving software evolution. In this thesis, 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 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 synthesiz-

ing relevant content as it emerges in the discussion. We benchmark models for generating solution descriptions and design a classifier for determining when sufficient context for generating an informative description becomes available. We investigate approaches for real-time generation, entailing separately trained and jointly trained classification and generation models. Furthermore, we also study techniques for deriving natural language context from bug report discussions and generated solution descriptions to guide models in generating suggested bug-resolving code changes.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software is constantly evolving as developers make changes to refactor code, support new functionality, and fix bugs. When software projects are developed collectively 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 making 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 aim to *support* software evolution by upholding software quality amidst constant code changes and *drive* software evolution by expediting critical code changes. We address these two goals through natural language.

Natural language serves as an important medium for search, documentation, and communication throughout the software development process. For example, 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 to document key aspects of the code. When a software bug is found, a bug report is opened, 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. In this thesis, we present novel tasks, datasets, and machine learning models which leverage natural language to facilitate software evolution.

For our first goal of supporting software evolution by 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 software evolution, specifically for expediting critical code changes that resolve 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 also study a classification task for determining when this context becomes available and develop pipelined and jointly trained approaches for supporting a real-time generation system. Additionally, we leverage bug report discussions and generated

solution descriptions to guide automated bug-fixing models in generating suggested code changes for bug resolution (Chapter 8).

Finally, we conclude by summarizing our main contributions (Chapter 10) and outlining directions for future work (Chapter 9).

Chapter 2

Background and Related Work

In this chapter, we discuss background information on relevant topics and prior work related to the research that we present in this thesis. We begin with an overview of software evolution (Section 2.1) followed by a high-level summary of common software-related tasks that use natural language (Section 2.2). We then describe the two specific forms of natural language that are the focus of this thesis: source code comments (Section 2.3) and bug report discussions (Section 2.4). Finally, we outline predominant techniques used to model (Section 2.5) and process (Section 2.6) software-related data.

2.1 Software Evolution

To quickly deliver software to users, software teams 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 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., 2019), 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.

2.2 Natural Language for Software-Related Tasks

There is growing interest in software-related tasks that combine various forms of natural language (NL) with source code. Many have studied generating code for a given NL input (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).

Some have explored code search based on NL queries (Husain et al., 2019; Cambronero et al., 2019; Zhao and Sun, 2020). Prior work examines tasks for generating natural language commit messages (Loyola et al., 2017; Xu et al., 2019a), release notes (Moreno et al., 2014), 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.

There has been very limited work in building interactive dialogue-based 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; Austin et al., 2021) 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 user intent.

2.3 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, 2016a) 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. Additionally, Movshovitz-Attias and Cohen (2015) develop an approach for detecting coordinate relationships between nouns in technical text on StackOverflow by grounding them to Java classes. 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.

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. In Chapter 4, we also study inconsistency detection. Unlike prior work, we develop a general-purpose, machine learning approach that is not catered towards any specific types of inconsistencies or comments. We instead capture deeper code/comment relationships by learning their syntactic and semantic structures. More importantly, in contrast to all of these listed approaches which detect inconsistencies that already exist in a code base, we aim to detect inconsistencies immediately upon code changes, before they are merged into the code base.

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. Our approach does not require such extensive feature engineering. Although their task is slightly different, we consider their approach as a baseline. 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. Our model instead leverages the full method context and also learns to map the comment directly to the code changes. 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.

2.4 Bug Report Discussions

Software bugs in open-source projects are reported through 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).

Many tasks have been proposed 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; Koyuncu et al., 2019). 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), dialogue act classification (Enayet and Sukthankar, 2020), and summarization (Rastkar et al., 2014; Jiang et al., 2017; Li et al., 2018b; Liu et al., 2020).

In Chapter 7, we propose a task for generating a natural language solution description by synthesizing content relevant to the solution from within the bug report discussion, as soon as the relevant context becomes available in the ongoing discussion. While this shares some similarities with bug report summarization, we identify some major differences. Approaches for bug report summarization are designed to generate holistic summaries of bug reports, with a summary being 25% of the length of the bug report (Liu et al., 2020). We instead aim to generate a concise description that captures a specific aspect of the bug report, namely the content related to the solution. Next, bug report summaries are not widely available, so approaches for this task rely on unsupervised techniques (Li et al., 2018b; Liu et al., 2020) or supervision from a small amount of data (Rastkar et al., 2014; Jiang et al., 2017). On the other hand, our approach for obtaining noisy supervision (Section 7.2) allows us to train supervised models on a large amount of data. Moreover, bug report summarization is a post hoc task, done after the bug has been resolved, to help developers address related bug reports in the future. In contrast, our goal is to help resolve the present bug report, so our system must learn *when* to perform generation during an ongoing discussion. Approaches for bug report summarization have been predominantly extractive whereas ours is abstractive.

2.5 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., 2019; 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., 2019a; 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.

For tasks entailing code edits, prior work has put forth various types of edit representations, including token-level and line-level sequential representations, AST-based representations, as well as graph-based representations (Shin et al., 2018; Yin et al., 2019; Chakraborty et al., 2020; Tao et al., 2021). In Section 4.2, we designed a span-level sequential edit representation as well as a consolidated AST-based edit representation which we found to better correlate with comments and comment edits, for the tasks we study in Chapters 4-6.

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) and CodeT5 (Wang et al., 2021) are 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. Chen et al. (2021a) fine-tuned GPT-3 (Brown et al., 2020) on data from millions of GitHub repos to build Codex, which powers GitHub Copilot[1]. Chowdhery et al. (2022) developed PaLM, a large pretrained language model that achieves state-of-the-art performance through few-shot learning on hundreds of tasks, including code understanding and generation tasks. Furthermore, to help these large models better reason about complex programs, some have proposed breaking down problems into multiple steps (Nye et al., 2021; Nijkamp et al., 2022).

In Chapters 7-8, we explore how such large models can be applied on new tasks and new forms of data in this domain by leveraging PLBART for learning from bug report discussions.

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

2.6 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; Lee et al., 2022), 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 on manually annotated subsets of data for this purpose. Others used task-specific heuristics (Allamanis et al., 2016; Hu et al., 2018; Fernandes et al., 2019; Allamanis, 2019). For example, Allamanis et al. (2016) discard overridden methods for the method naming task due to them having repetitive names.

Similar to prior work, we also find it necessary to filter data to handle noise for all the various datasets we collect as part of this thesis. We design filtering heuristics specific to the tasks we study (Sections 3.2, 4.3, 7.2).

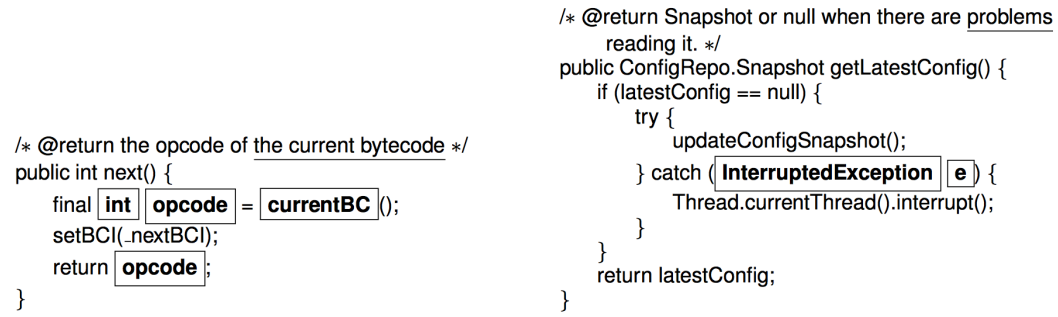
Chapter 3

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). This chapter is based on work that was presented 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



(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.

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 mul-

tiple, 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 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 $\sim$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.

[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.

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.

Additions: 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 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. This set of examples we collect from additions constitute our *primary dataset*.

Deletions: If we assume that the code tokens in deleted lines are associated with an NP that is deleted from the comment, we can extract one more example from each commit. However, deleted NPs are much more subtle in this respect than added NPs. As stated above, since the added NP does not exist in the previous version, it is unlikely that code tokens in lines that existed previously are associated with it. On the other hand, since the deleted NP does exist in the previous version, we cannot reliably claim that a code token in a line that is unchanged between versions is not associated with the NP. This could consequently lead to more noise for the negative label in addition to the noise that inherently exists for the positive label. For instance, in the deleted example from Figure 3.2, `_nextBCI` is automatically labeled as not associated with the deleted NP "the next bytecode" even though it

is arguably associated. Hence we separate such examples from our primary dataset and form another set of examples we refer to as *the deletions dataset*.

3.2.2 Processing

We examine the two versions of the code and comment in a commit: *before* commit and *after* commit. Using spaCy[2], we extract NPs from the two versions of the comment, and using the javalang library, we tokenize the two versions of the code. Using the difflib library, we compute the *diff* between the NPs in the two versions of the comment as well as the diff between the two versions of the tokenized code sequences. These *diff*s are marked with plus and minus signs for each changed line, as shown in Figure 3.2.

From the diffs, we identify the NPs and code tokens that are unique to either the *before* or the *after* versions of the comment and code respectively, allowing us to construct two pairs in the form *(NPs, associated code tokens)*. If either the extracted NPs or list of associated code tokens is empty, we discard the pair. Additionally, we discard pairs consisting of more than one NP to obtain unambiguous training data for determining which code tokens should be associated with which NP. Therefore, the final set of pairs are in the form *(NP, associated code tokens)*. Note that for any token in the associated code tokens, if it is not a common Java type (e.g., `int`, `String`), we also treat any other token in the sequence with the same literal string as associated.

We then go back to the *before* and *after* versions of the code (excluding

[2] `https://spacy.io/`

Java keywords, operators and symbols, c.f. Section 3.1). We tokenize the code sequence 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. The example extracted from the previous version (*before*) is added to the deletions dataset and the one from the new version (*after*) is added to the primary (additions) dataset.

3.2.3 Filtering

We apply heuristics to reduce noise, as done in prior work (Section 2.6). We impose constraints to filter out duplicates, trivial cases, and examples consisting of code and comment changes that are unrelated. We define trivial cases as those examples involving single-line methods which consist of only a few code tokens that are all likely to be associated with the NP as well as those examples in which all associations can be resolved with a simple string matching tool.

Additionally, after manually inspecting a sample of approximately 200 examples, we establish heuristics to minimize the number of examples with unrelated code and comment changes: (1) those that have lengthy methods or a substantial number of code changes which are likely not to all be correlated with the comment; (2) cases with changes to the code and comment that are related to re-formatting, typo fixes, and simple rephrasing; (3) examples involving comment changes entailing verb phrases as the corresponding code changes could be related to these phrases rather than the NP. In addition, since we focus on the `@return` tag that serves to describe the return value of a Java method, we eliminate examples with

	Examples	Candidate Code Tokens		
		Total	Unique	Average
Train	776	23,188	5,908	29.9
Valid	77	2,488	911	32.3
Test (annotated)	117	3,592	1,266	30.7
Deletions	867	25,203	6,186	29.1

Table 3.1: Number of examples, total and unique candidate tokens, and average number of candidate tokens per example, for each partition of the dataset.

code changes that do not include either a change to the return type or at least one return statement.[3]

Applying such heuristics substantially reduced the size of our dataset. However, we determined such filtering to be necessary after manually inspecting 200 examples and observing significant noise, and finding that is consistent with aforementioned prior work, which pointed out that the levels of noise in large code bases is too substantial to learn from without aggressive filtering and pre-processing.

3.2.4 Dataset Statistics

Upon filtering, we partition our primary dataset into train, test, and validation sets, shown in Table 3.1. For more reliable evaluation, the 117 examples in the test set are annotated by one of the authors who has 7 years of experience with Java. Based on the assumption that there is minimal noise in the code tokens automatically labeled as not associated, only the tokens that are automatically labeled as associated are re-labeled as associated or not associated.[4] Furthermore, since we conduct some experiments involving training data from the deletions dataset, we

[3] See Section 3.6.2 for more details.

[4] See Section 3.6.3 for examples of annotations.

filter out any example from the deletions dataset that is extracted from the same commit as an example from the test set. Based on the training set, the median number of words in the NP is 2 with an interquartile range (IQR, difference between 25% and 75% percentile)[5] of 1, the median number of code tokens is 25 with IQR 21, and the median number of associated code tokens is 10 with IQR 13. Our dataset is publicly available.[6]

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). *Subtokenization* refers to splitting a conjoined code token into its subtokens (e.g., camelCase $\rightarrow$ camel, case; snake_case $\rightarrow$ snake, 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.

[5]We report IQR since the distributions are not normal.

[6]`https://github.com/panthap2/AssociatingNLCommentCodeEntities`

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 SEQ2SEQ model (Sutskever et al., 2014). This allows us to learn embeddings which capture aspects of comments and source code, which are likely not captured in other types of embeddings that are pre-trained on only natural language text (e.g., BERT (Devlin et al., 2019)).[7] 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 using the embeddings trained on Java code. We then com-

[7]Pretrained models like CodeBERT (Feng et al., 2020) were not available at the time of this work, though we believe embeddings from such models could be useful here.

pute the features as the values corresponding to 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. Using javalang's AST parser, we derive the AST corresponding to the method. 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 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 feed-forward 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 $\sim$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.

	Annotated			Unannotated		
Model	**Precision**	**Recall**	**F1**	**Precision**	**Recall**	**F1**
Random	32.1	47.2	38.2	39.6	49.8	44.1
Weighted random	33.8	42.8	37.8	39.5	42.5	40.9
Subtoken matching	56.7	33.8	42.8	58.3	29.4	39.1
Presence in return line	51.5	45.8	48.5	56.1	42.3	48.2
Binary Classifier	**57.4**	65.4	**61.0**	**64.7**	**63.3**	**64.0**
CRF	48.4	**66.3**	55.9	52.1	58.1	53.3

Table 3.2: Micro precision, recall, and F1 scores after training on the primary training set, evaluated on the annotated and unannotated test sets. Differences between F1 scores within the same test set are statistically significant based on a signed rank t-test, with $p < 0.01$.

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

We evaluate our models using micro-level precision, recall, and F1 metrics. That is, we evaluate our models at the token-level, on the 3,592 NP-code token pairs in the test set. All reported scores are averaged across three runs. In the following sections, we discuss results from training on just the primary training set, results from incorporating the deletions dataset into training, and results from an ablation study of the features used by the binary classifier and CRF model.

3.5.1 Training on Primary Dataset

The results of the three baselines and our models are given in Table 3.2. Our analysis is primarily based on the results on the annotated test set, and we show the

results from the unannotated set simply for completeness. Relative to scores from the unannotated set, the models tend to achieve lower precision scores and higher recall scores with the annotated set. This is expected since the number of tokens with the gold label *associated* was reduced during the annotation procedure.

Both of our models outperform the baselines by wide margins. See Section 3.6.4 for sample output from the binary classifier. 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.

	Binary Classifier			CRF		
# of Deletion Examples	**Precision**	**Recall**	**F1**	**Precision**	**Recall**	**F1**
0	57.4	65.4	61.0	48.4	66.3	55.9
100	57.2	63.9	60.3	48.2	73.6	58.2
200	55.4	68.9	61.4	51.2	68.5	58.5
500	62.4	69.3	65.5	50.4	74.0	59.9
867	**64.4**	**71.5**	**67.7**	**52.8**	**74.5**	**61.8**

Table 3.3: Micro precision, recall, and F1 scores after training on the primary training set and a varying number of deleted examples, tested on the annotated test set.

3.5.2 Augmenting Training with Deletions

We increase the training set by adding data in stages from the deletions dataset. The results from training the binary classifier and CRF on these new supplemented datasets are shown in Table 3.3. For the binary classifier, adding 500 and 867 deleted examples seems to provide a significant boost in F1, and for the CRF model, adding any amount of deleted examples leads to improved performance. This indicates that our models can learn from data that we consider to be more noisy than the primary training set that we collect. Since we are able to find value in both the added case as well as the deleted case corresponding to a given commit, we are able to substantially increase the upper bound on the amount of data that can be collected to train models that perform our proposed task. This is particularly encouraging given how difficult it is to obtain a large amount of high-quality data for this task. Despite having extracted examples from methods in source code files across all commits of more than 1,000 projects, we only acquire a total of 970 examples from added cases after filtering for noise. By including the 867 examples from deleted cases, we increase this number to 1,837. While this is still a relatively

Model	Precision	Recall	F1
Full	57.4	65.4	**61.0**
- code embeddings	51.9	61.7	56.2
- comment embeddings	52.3	**67.5**	58.7
- cosine similarity	**58.2**	61.3	59.7
- Java API & AST	54.3	64.1	58.8

Table 3.4: Micro precision, recall, and F1 scores for the binary classifier upon ablating certain features, tested on the annotated test set. All differences in F1 are statistically significant based on a signed rank t-test, with $p < 0.01$.

small number, we expect the potential size to increase substantially as the scope of the task is extended to other comments beyond the CodeIn@return comments that we focus on in this paper for an initial study.

3.5.3 Ablation Study

We conduct an ablation study on the binary classifier trained on the primary dataset in order to analyze the impact of the features we introduce. We ablate cosine similarity, embedding, and the Java-related features. The embedding features include code embeddings (i.e., the embeddings corresponding to the candidate code token and the tokens in the line of the method) and comment embeddings (i.e., the embeddings corresponding to the NP and CodeIn@return comment). [8] Based on the results shown in Table 3.4, all of these features contribute in a positive manner towards the performance of the full model, with respect to the F1 metric.

[8]Models without embeddings also do not include cosine similarity, as the latter depends on the embeddings.

Category	Filter	# Discarded
Trivial	Short methods	5,218
	Lexical string matching	828
Unrelated	No return statement/type change	3,709
	Long methods	692
	Many added code tokens	67
	Many diff lines	6
	Added VP	397
	Re-formatting/typos/rephrasing	1,275
Other	Duplicates	546
	Multiple NPs added	1,574
	No NPs added	856
	No code tokens added	167
		15,335

Table 3.5: Number of examples filtered out of the primary dataset by each heuristic. Prior to filtering, there are 16,305 examples, and following filtering, there are 970 examples.

3.6 Additional Details

3.6.1 Model Parameters

The 4 fully-connected layers have 512, 384, 256, and 128 units. Dropout is applied to each of these with probability 0.2. We terminate training if there is no improvement in the F1 score on the validation set for 5 consecutive epochs (after 10 epochs), and we use the model corresponding to the highest validation F1 score up till that point. We implemented both models with TensorFlow.

3.6.2 Filtering Details

The number of examples filtered out from the primary dataset by each heuristic is shown in Table 3.5. In this section, we discuss specific parameters used for

many of these filtering cases.

Poorly maintained projects We extract the majority of examples from the top 1,000 projects in order to minimize the use of poorly maintained projects that may have inconsistent code and comments as a result of developers not updating comments when making code changes (Jiang and Hassan, 2006).

Trivial cases We do not consider methods with less than 4 lines of code as we observe that such methods generally have only one line in the method body, and we believe it would be trivial to classify the few code tokens present in such a method. Additionally, we remove cases in which the lexical string of *all* the associated code tokens match some component of the given NP in the comment, either at the token-level or *subtoken*-level (e.g., for NP "max result" and code token `maxResult`).

Unrelated code and comment changes Because we are focusing on the `@return` tag of the Javadoc comment, code changes involving a return statement or return type are more likely to be relevant to the change in the comment, and so we only consider examples extracted from commits in which the code change also includes either a change to at least one return statement or the return type of the method. Furthermore, we discard examples involving methods that are longer than 30 lines, which is the 90th percentile for method lengths in the original primary dataset that we collect. Since it is unlikely that a `@return` comments can capture the essence of extremely long methods, we eliminate such cases. Moreover, most coding standards discourage such long methods, suggesting that these methods could be poorly written and possibly even poorly maintained. Additionally, the 90th percentile for the number of associated code tokens is approximately 40, and in order to reduce

the number of cases in which there are substantial code changes that may be unrelated to the change in the comment, we remove such examples from our dataset. We also eliminate examples extracted from diffs involving more than 500 lines for a similar reason. To add, we disregard examples in which there are changes involving verb phrases in the comment as the code changes could be related to these phrases rather than the NP. We impose constraints to limit commits that involve insubstantial changes to the code or comment such as re-formatting, typo fixes, and simple rephrasing.

3.6.3 Annotation Examples

```
/* @return a String representing
      the current active version of the ConfigStore.*/
@Override
public String getCurrentVersion() {
    try {
        return storeMetadata.getCurrentVersion();
        } catch (IOException e) {
            Path configStoreDir = new Path(new
                Path(this.physicalStoreRoot),
                CONFIG_STORE_NAME);
            throw new RuntimeException(String.
                format("Error while checking
                current version for configStoreDir
                : \"%s\"", configStoreDir), e);
        }
    }
```

(a) eclipse-egit example

```
/* @return the modified content*/
public byte[] getModifiedContent() {
  byte[] result = new byte[modifiedContent.length];
  System.arraycopy(modifiedContent, 0, result, 0,
      modifiedContent.length);
  return result;
}
```

(b) apache-incubator-gobblin example

Figure 3.3: Annotation example with all the bolded code tokens being automatically labeled as associated with the underlined NP in the comment and the crossed out tokens being manually re-labeled as not associated.

We illustrate our annotation procedure in Figure 3.3. We consider only code tokens that are automatically labeled as associated and re-label any of these that we find to be irrelevant to the specified NP as not associated.

3.6.4 Sample Output

```
/* @return the SaveStepExecutionRes*/
private SaveStepExecutionRes
        handleSaveStepExecution(
        SaveStepExecutionReq request) {
    SaveStepExecutionRes response = null;
    try {
        StepExecution stepExecution =
                JobRepositoryRpcFactory.
                convertStepExecutionType(request.
                stepExecution);
        stepExecutionDao.saveStepExecution(
                stepExecution);
        response = new SaveStepExecutionRes(
                stepExecution.getId(), stepExecution.
                getVersion());
        } catch (Exception e) {
            log.error("error handling command", e);
        }
        return response;
    }
```

(a) spring-yarn example

```
/* @return The "advance" value or 0 if there is no text.*/
private int getTextWidth(TextLayout textLayout) {
    if (textLayout != null) {
        return (int)Math.ceil(
            textLayout.getAdvance());
    }
    return 0;
}
```

(b) apache-pivot example

Figure 3.4: Sample output of the binary classifier. The model classifies the bolded code tokens as associated with the underlined NP in the comment. The manually labeled, gold code tokens that are associated with the NP are in blue.

We provide sample output from the binary classifier in Figure 3.4. In all examples, the bolded code tokens denote the tokens that the model predicts to be associated with the underlined NP in the comment and the highlighted ones indicate the true code tokens that the NP is associated with, as determined by manual annotation. For the example shown in Figure 3.4a, the model's predictions matches the gold associations in the annotated test set. In Figure 3.4b, the classifier accurately classifies the code token that is truly associated with the NP; however, it incorrectly identifies `Math` and `0` to be associated with the NP. This could be because the word "value" is often correlated with mathematical operations and numerical values, and so it could have possibly appeared in the training data with code tokens such as

`Math` and `0`.

3.7 Summary

In this work, we formulated the task of associating entities in comments with elements in source code. We proposed a novel approach for obtaining noisy supervision for this task, and we presented a rich set of features that aim to capture aspects of the code, comments, and the relations that hold between them. Based on evaluation conducted on a manually labeled test set, we showed that two different models trained on such noisy data can significantly outperform multiple baselines. Moreover, we demonstrated the potential for learning from noisy data by showing how increasing the size of the noisy training data can lead to improved performance. We also highlighted the value of our feature set through an ablation study.

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 2.3). 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 this chapter, 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. This chapter is based on work originally presented in Panthaplackel et

```
/**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.

al. (2021).

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

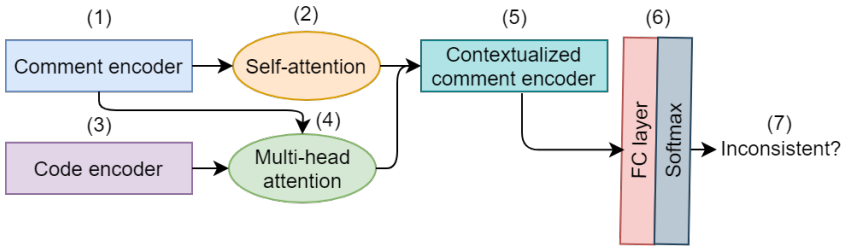


Figure 4.2: High-level architecture of our approach for inconsistency detection.

array. When the method is modified to 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.

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.

[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).

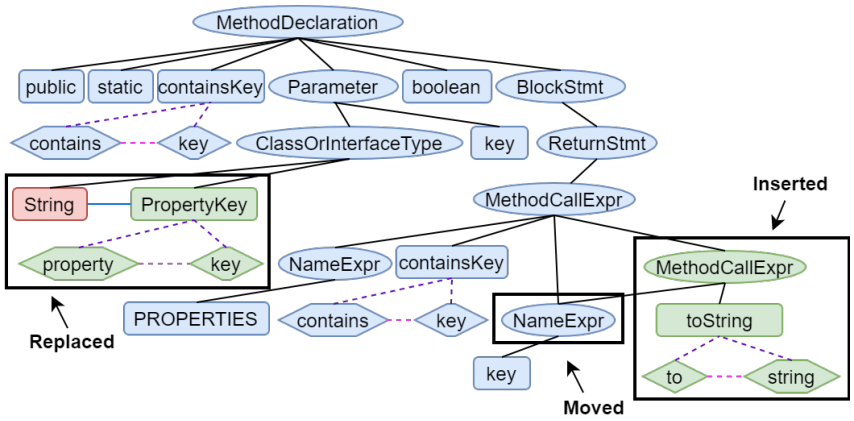


Figure 4.3: AST-based code edit representation (M_{edit}) corresponding to Figure 4.1b, with removed nodes in red and added nodes in green.

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` $\rightarrow$ `MethodDeclaration`) and *child* (e.g., `MethodDeclaration` $\rightarrow$ `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 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.

	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: Dataset partitions for inconsistency detection

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 2.6). 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,

	@return	@param	Summary	Full
C_{old}	9.7	8.4	13.3	10.3
M_{old}	131.1	186.9	137.0	147.2
M_{new}	131.9	187.7	135.4	147.3
M_{edit}	179.4	240.9	186.6	197.3
T_{old}	127.2	184.1	130.5	142.9
T	128.1	184.5	129.5	143.2
T_{edit}	154.3	213.7	159.1	171.1

Table 4.2: Statistics on the average lengths of comment and code representations for inconsistency detection.

to construct a balanced dataset. Partition sizes 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.[2] Note that we subtokenize M_{new}, and M_{edit} (as described previously in Section 3.3). Since comments often include code tokens, we also subtokenize C_{old}.

In Table 4.2, we show the average lengths of comment and code representations for the various types of comments in our dataset. The lengths for C_{old} and sequential code representations (i.e., M_{old}, M_{new}, M_{edit}) are computed based on the subtokenized sequences that are used by our model. Note that the M_{edit} representation also includes edit keywords. We report the sizes of the AST representations (T_{old}, T, T_{edit}) in terms of number of nodes. This also includes the added *subnodes*. Our dataset is publicly available.[3]

[2]See Section 4.6.2 for additional details about data filtering and annotation.

[3]`https://github.com/panthap2/deep-jit-inconsistency-detection`

4.4 Models

In the following section, 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, OVERLAP(C_{old}, 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 (BOW) 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. SEQ(C_{old}, M_{new}) encodes M_{new} with a GRU, GRAPH(C_{old}, T) encodes T with a GGNN, and HYBRID(C_{old}, M_{new}, T) uses both. Multi-head attention in HYBRID(C_{old}, M_{new}, T) is computed with the hidden states of the two encoders separately and then combined.[4]

[4]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 adja-

	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 4.3: 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 ($p < 0.05$).

cent tokens that are lost once the body of code and comment are transformed into single, subtokenized sequences.

4.5 Results

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 4.3, 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 OVERLAP(C_{old}, deleted) 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, we analyze the performance of the three just-in-time + features models with respect to individual comment types:

Evaluating `@return` Comments:

We train the (learned) baselines introduced in Section 4.4.1 on only the 15,950 examples pertaining to `@return` comments. We additionally consider two

	Cleaned Test Sample				Full Test Set			
	P	**R**	**F1**	**Acc**	**P**	**R**	**F1**	**Acc**
Training: `@return` Only								
OVERLAP(C_{old}, deleted)	69.2	54.0	60.7	65.0$^{\|}$	67.6$^{\|\P}$	53.3$^{\|}$	59.6	63.9
Corazza et al. (2018)	73.2	60.0¶	65.9	69.0	68.9¶	61.2§	64.8	66.8
CodeBERT BOW (C_{old}, M_{new})	84.9*	74.7§	79.4*¶	80.7*	85.6	82.7	**84.1**	**84.3**
CodeBERT BOW (C_{old}, M_{edit})	62.5	74.0§	67.7$^{\|}$	64.7$^{\|}$	66.8$^{\|}$	78.8*†	72.2	69.7
Liu et al. (2018b)	76.9	62.0¶	68.6$^{\|}$	71.7	76.0	63.0§	68.9	71.6
Khamis et al. (2010)	52.1	**98.0**	68.1$^{\|}$	54.0	51.6	**97.3**	67.4	52.9
GENMATCH	64.6	62.0¶	63.3	64.0$^{\|}$	60.4	54.9$^{\|}$	57.5	59.5
SEQ(C_{old}, M_{edit}) + features	85.3*	75.3§	79.9¶	81.0*	87.2§	75.9	81.2*§	82.4
GRAPH(C_{old}, T_{edit}) + features	87.4	77.3*	**82.0**	**83.0**	84.0*	78.0*	80.8*§	81.4*†
HYBRID(C_{old}, M_{edit}, T_{edit}) + features	84.8*	78.0†	81.2	82.0¶	84.3*	78.7†	81.3§	81.9*
Training: Combined								
SEQ(C_{old}, M_{edit}) + features	88.5§	72.0$^{\|}$	79.4*¶	81.3*¶	**87.6**§	73.3¶	79.8¶	81.4*†
GRAPH(C_{old}, T_{edit}) + features	81.2	77.3*†	79.1*	79.7	82.2	79.3†	80.6*	80.9†
HYBRID(C_{old}, M_{edit}, T_{edit}) + features	**88.7**§	72.0$^{\|}$	79.4*	81.3*	87.3§	73.7¶	79.8¶	81.4*†

Table 4.4: Results for `@return` examples. Scores for which the difference in performance is *not* statistically significant ($p < 0.05$) are shown with identical symbols.

baselines for `@return` comments. Khamis et al. (2010) proposed a heuristic for detecting inconsistency in `@return` comments: the comment must begin with the correct return type of the corresponding method. We implement a baseline based on this heuristic. We also remove articles (e.g., *a*, *the*) from the beginning of the comment before applying this rule, as we found this to improve performance. We introduce another baseline, GENMATCH, in which we use a comment generation model[5] to generate an `@return` comment for M_{old} and an `@return` comment for M_{new}. If the two comments match exactly, we consider the code change to be irrelevant to `@return` comments and thus the existing `@return` comment remains consistent. We compare these baselines with our models, trained on only `@return` comments. We additionally compare with our models, trained on the combined training set, as done in the main paper.

In Table 4.4, we report results on the 100 `@return` examples in the cleaned

[5]We rely on the comment generation model used to build embeddings in Section 3.3

test set as well as the 1,840 @return examples in the full test set. While the CodeBERT BOW (C_{old}, M_{edit}) baseline performs quite well here, our approach can outperform baselines (w.r.t. F1 and Acc) on the cleaned test sample, when trained on only @return comments. We find that training on the combined dataset slightly deteriorates performance of our models. This is not surprising as in combined training, models must learn to generalize across comment types, not just @return comments. Nonetheless, the difference in performance between training on the comment-specific and combined sets are relatively small.

Evaluating @param Comments:

	Cleaned Test Sample				Full Test Set			
	P	**R**	**F1**	**Acc**	**P**	**R**	**F1**	**Acc**
Training: @param Only								
OVERLAP(C_{old}, deleted)	85.7	**96.0***§	90.6	90.0	84.0	**93.3**	88.4¶	87.8¶
Corazza et al. (2018)	74.1	40.0	51.9	63.0	59.1§	43.9	50.4	56.7
CodeBERT BOW (C_{old}, M_{new})	62.8	57.3	59.9	61.7	58.9§	64.4	61.5	59.7
CodeBERT BOW (C_{old}, M_{edit})	81.8	84.0	82.8	82.7	75.5	82.7	78.9§	77.9
Liu et al. (2018b)	90.4§	62.7	74.0	78.0	88.6¶	72.3	79.6§	81.5
Khamis et al. (2010)	**97.8***	90.0$^{\|}$	93.8	94.0†	87.7¶	89.0*§	88.3¶	88.2¶
SEQ(C_{old}, M_{edit}) + features	95.4	**96.0***	95.7*†	95.7*	91.4	89.2§	90.3†	90.4†§
GRAPH(C_{old}, T_{edit}) + features	97.3*	94.0	95.6*	95.7*	**94.9***	90.0	**92.4**	**92.6**
HYBRID(C_{old}, M_{edit}, T_{edit}) + features	96.6†	95.3§	**96.0**†	**96.0**	94.3†	89.3§	91.7*	91.9*
Training: Combined								
SEQ(C_{old}, M_{edit}) + features	90.0§	95.3§	92.5	92.3§	92.2	88.3*	90.2†	90.4†
GRAPH(C_{old}, T_{edit}) + features	96.5†	92.0	94.2	94.3†	94.5*†	89.0*§	91.7*	91.9*
HYBRID(C_{old}, M_{edit}, T_{edit}) + features	94.6	89.3$^{\|}$	91.8	92.0§	93.3	85.9	89.4	89.9§

Table 4.5: Results for @param examples. Scores for which the difference in performance is *not* statistically significant ($p < 0.05$) are shown with identical symbols.

For @param comments, we consider another baseline designed to follow the heuristic proposed by Khamis et al. (2010) for this comment type: the comment should begin with the name of the parameter being documented. We remove articles from the beginning of the comment and consider whether the first term is one

of the arguments of the method. If this is not the case, we classify it as inconsistent. We consider the comment-specific and combined settings, as we do for `@return` comments. In Table 4.5, we report results on the 100 `@param` examples in the cleaned test set as well as the 1,038 `@param` examples in the full test set. We find that rule-based baselines can perform very well for `@param` comments, especially the Khamis et al. (2010) baseline. This suggests that most `@param` comments conform to the format they suggested. Nonetheless, our models are able to *learn* this without explicitly specifying this format and can even achieve higher performance (by statistically significant margins) when trained on only `@param` comments. The combined setting slightly deteriorates performance of our models; however, the GRAPH(C_{old}, T_{edit}) + features model can still perform slightly better than Khamis et al. (2010) w.r.t. F1 on the cleaned test sample.

Evaluating Summary Comments:

	Cleaned Test Sample				Full Test Set			
	P	**R**	**F1**	**Acc**	**P**	**R**	**F1**	**Acc**
Training: Summary Only								
OVERLAP(C_{old}, deleted)	75.0*	66.0	70.2§	72.0	71.4	49.7	58.6	64.9§
Corazza et al. (2018)	55.6	30.0	39.0	53.0	61.7	41.1	49.3	57.8
CodeBERT BOW (C_{old}, M_{new})	61.6†	78.7†	68.8	64.3	63.7§	**75.6***	68.9*†§	66.0§†
CodeBERT BOW (C_{old}, M_{edit})	62.1†	80.0§	69.8§	65.3	64.5§	72.4§	68.0†§	65.9§†
Liu et al. (2018b) 2018b	85.2	76.7	80.7†	81.7	77.1*	57.0†¶	65.5	70.0
SEQ(C_{old}, M_{edit}) + features	72.7	**92.7**	81.4†	78.7	67.7	74.3*	**70.6**‖	68.9
GRAPH(C_{old}, T_{edit}) + features	74.3*	92.0*	82.0	79.3	68.4	70.9	69.2*	68.2
HYBRID(C_{old}, M_{edit}, T_{edit}) + features	70.7	90.0	79.2	76.3	64.5§	72.9§	68.4†§	66.3†
Training: Combined								
SEQ(C_{old}, M_{edit}) + features	**96.0**	78.7†§	86.5*	87.7	84.7†	58.3†	69.0*†	**73.9***
GRAPH(C_{old}, T_{edit}) + features	80.8	92.0*	86.0*	85.0	76.0*	66.4	**70.6**‖	72.5
HYBRID(C_{old}, M_{edit}, T_{edit}) + features	93.7	86.0	**89.5**	**90.0**	**85.0**†	57.0¶	68.1§	73.5*

Table 4.6: Results for summary comment examples. Scores for which the difference in performance is *not* statistically significant ($p < 0.05$) are shown with identical symbols.

Summary comments do not have a well-defined structure, and thus we do not have a format-based baseline as we did for `@return` and `@param` comments. We evaluate baselines and our models, trained on comment-specific data, as well as our model trained on the combined training set. In Table 4.6, we report results on the 100 summary examples in the cleaned test set as well as the 1,066 summary examples in the full test set. While Liu et al. (2018b) is a strong baseline here, we find that we can outperform all baselines in the combined training setting. Unlike the case for `@return` and `@param` comments, combined training appears to yield improved performance over comment-specific training for our models. This suggests that the models can extract valuable information from the more structured comments in the training set that pertain to specific parts of the code in order to address the less-structured summary comments.

4.6 Additional Details

4.6.1 Model Parameters

Models are trained to minimize negative log likelihood. We use 2-layer bi-GRU encoders (hidden dimension 64). GGNN encoders (hidden dimension 64) are rolled out for 8 message-passing steps, also use hidden dimension 64. We initialize comment and code embeddings, of dimension 64, with pretrained ones (Section 3.3). Edit embeddings are of dimension 8. Attention modules use 4 attention heads. We use a dropout rate of 0.6. Training ends if the validation F1 does not improve for 10 epochs.

4.6.2 More Data Details

We provide additional information about our procedures for filtering the dataset and curating a sample of the test set for evaluation. We also include various statistics about the examples that comprise our dataset.

Filtering

Recall from our data collection procedure (§4.3) that we extracted comment/method pairs from consecutive commits, of the form (C_1, M_1), (C_2, M_2), where C_{old}=C_1, M_{old}=M_1, and M_{new}=M_2. We apply heuristics to reduce the number of cases in which there are unrelated comment and code changes. We filter out positive examples in which the differences between C_1 and C_2 entail minor cosmetic edits (e.g., reformatting, spelling corrections). As done previously in Section 3.6.2, for `@return` examples, we require there to be a code change to at least one return statement or the return type of the method. We discard all `@return` examples (positive and negative) that do not satisfy this condition. This is because `@return` comments describe aspects of the return value of a method, which is typically related to the method return type and return statements. We apply the same constraint to summary comment examples, since they often describe aspects of the output (e.g., Figure 4.1a). Because `@param` comments generally pertain to the method's arguments, we only use examples in which an argument name or type is changed within the method. Next, we reduce the number of noisy negative examples in which a developer fails to update comments in accordance with code changes by avoiding poorly maintained projects (Section 3.6.2). Furthermore, because we con-

	Positive	Negative	Total
`@return`	9,807	72,826	82,633
`@param`	5,507	19,007	24,514
Summary	5,904	69,650	75,554
Full	21,218	161,483	182,701

Table 4.7: Dataset sizes before downsampling for inconsistency detection.

sider AST representations, we remove all examples consisting of a method which cannot be parsed into an AST. Additionally, we remove duplicate examples, as they have been found to negatively affect training machine learning models for source code (Allamanis, 2019).

Downsampling Negative Class

In our data collection procedure, we obtained many more negative examples. Because a naïve classifier trained to always predict the negative label can achieve high accuracy in such a setting, we downsample the negative class in order to obtain a balanced dataset. We provide the sizes of the positive and negative classes before downsampling in Table 4.7. Note that we also discard some examples to ensure no overlap between the projects in training, validation, and test.

Curating Test Sample

We construct a clean test set by randomly sampling without replacement from the full test set in such a way that we attain a balanced sample, in terms of both labels (i.e., positive, negative) and comment type (i.e., `@return`, `@param`, summary). We then remove mislabeled examples from this.

We remove 11% of examples for having the incorrect label, 3% for being uncertain about the correct label due to the limited context provided in the method, and 6% from being poor examples (e.g., comments like "document me" or code changes that simply comment out the entire method). Therefore, we find 17-20% noise. For the individual comment types, the percent of noise is 6-15% for `@return`, 14-16% for `@param`, and 26-28% for summary comments. For individual labels, it is 26-28% for positive and 8-12% for negative.

4.7 Summary

In this work, we developed a deep learning approach for just-in-time inconsistency detection between code and comments by learning to relate comments and code changes. Based on evaluation on a large corpus consisting of multiple types of comments, we showed that our model substantially outperforms various baselines as well as post hoc models that do not consider code changes.

Chapter 5

Updating Natural Language Comments Based on Code Changes

Once inconsistent comments are detected upon code changes (Chapter 4), 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 2.2). 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. This chapter is based on the work presented in Panthaplackel et al. (2020b).

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

(a) Previous

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

(b) Updated

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

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.

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[1] 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*

[1]We had also considered transformers (Vaswani et al., 2017); however, we found no advantage during preliminary experiments. Additionally, prior work (Fernandes et al., 2019) found that they yield lower performance for comment generation.

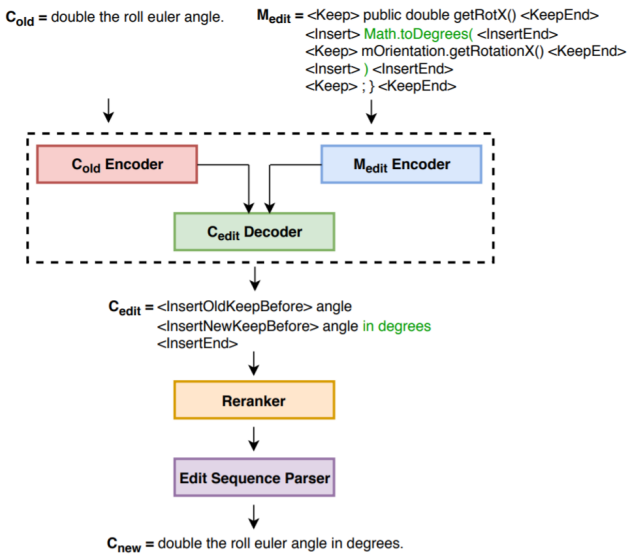


Figure 5.2: High-level architecture of our edit model for updating comments.

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. We provide more details on this procedure in Section 5.9.2.

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 model is 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

		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 Length	Unique	7,271	2,473	2,690
	Mean	86.4	87.4	97.4
	Median	46	49	50
Comment Length	Unique	4,823	1,695	1,737
	Mean	10.8	11.2	11.1
	Median	8	9	9

Table 5.1: Comment update dataset statistics. 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.

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.

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 Automatic Evaluation

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;[2] GLEU (Napoles et al., 2015), used in grammatical error cor-

[2]Although the original formulation only used precision for the delete operation, more recent

	xMatch (%)	METEOR	BLEU-4	SARI	GLEU
Baselines					
Copy	0.0	34.6	46.2	19.3	35.4
Return type subt.	13.7§	43.1¶	50.8‖	31.7	42.5*
Return type subst. + null	13.7§	43.4	**51.2**†	32.1	42.6*
Non-reranked models					
Generation	1.1	11.9	10.5	21.2	17.4
Edit	17.7	42.2¶	48.2	**46.4**	45.1
Reranked models					
Generation	2.1	18.2	18.9	25.6	22.7
Edit	**18.4**	**44.7**	50.7‖†	45.5	**46.1**

Table 5.2: Exact match, METEOR, BLEU-4, SARI, and GLEU scores. Differences that are *not* statistically significant ($p < 0.05$) are shown with identical symbols.

rection and style transfer, takes into account the source sentence and deviates from BLEU by giving more importance to n-grams that have been correctly changed.

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

work computes F1 for this as well (Dong et al., 2019; Alva-Manchego et al., 2019).

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.

5.6 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.

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 com-

ment. 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.

Table 5.3 presents the percentage of annotations (out of 500) for which users selected comment suggestions that were produced by each model. Using Krippen-

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 ($p < 0.05$).

dorff'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.

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 of these, we observe that in a large portion of these cases, even though the comment was actually updated in the data that we collected, the comment did not actually warrant an update based on the given code changes. The comment was updated for tangential reasons (e.g., comments that became inconsistent based on a previous set of code changes). 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 integrat-

```
/* @return item in given position */
public Complex getComplex(final int i) {
    return get(i);
}
```

(a) Previous

```
/* @return item in first position */
public Complex getComplex() {
    return get();
}
```

(b) Updated

Figure 5.3: Changes in the `getComplex` method and its corresponding `@return` comment between two subsequent commits of the eclipse-january project, available on GitHub.

ing 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.

5.7 Error Analysis

We find that our model performs poorly in cases requiring external knowledge and more context than that provided by the given method. For instance, correctly updating the comment shown in Figure 5.3 requires knowing that `get` returns the item in the first position if no argument is provided. Our model does not have access to this information, and it fails to generate a reasonable update: "`@return` complex in given position." On the other hand, the reranked generation model produces "`@return` the complex value" which is arguably reasonable for the given context. This suggests that incorporating more code context could be beneficial for both models. Furthermore, we find that our model tends to make more mistakes when it must reason about a large amount of code change between M_{old} and M_{new}, and we found that in many such cases, the output of the reranked generation model was better. This suggests that when there are substantial code changes, M_{new} effectively becomes a *new* method, and generating a comment from scratch

Inputs	Output	xM (%)	METEOR	BLEU-4	SARI	GLEU
C_{old}, M_{new}	C_{new}	5.7‡¶	29.3†	33.5§	28.0	30.0*
	C_{edit}	4.8$^{\ddagger *}$	33.8	43.3	35.5	38.0$^{\|}$
C_{old}, M_{old}, M_{new}	C_{new}	3.7*	18.7	20.1	23.9	22.0
	C_{edit}	5.2‡¶	34.9	44.0*	33.5	37.6$^{\|}$
C_{old}, M_{edit}	C_{new}	6.1¶	30.0†	34.2§	29.0	30.5*
	C_{edit}	**8.9**	**36.2**	**44.3***	**40.5**	**39.9**

Table 5.4: Exact match, METEOR, BLEU-4, SARI, and GLEU for various combinations of code input and target comment output configurations. Features and reranking are disabled for all models. Scores for which the difference in performance is *not* statistically significant ($p < 0.05$) are indicated with matching symbols.

may be more appropriate. Ensembling generation with our system through a regression model that predicts the extent of editing that is needed may lead to a more generalizable approach that can accommodate such cases.

5.8 Ablations

We empirically study the effect of training the network to encode explicit code edits and decode explicit comment edits. As discussed in Section 5.2, the edit model consists of two encoders, one that encodes C_{old} and another that encodes the code representation, M_{edit}. We conduct experiments in which the code representation instead consists of either (1) M_{new} or (2) both M_{old} and M_{new} (encoded separately and hidden states concatenated). Additionally, rather than having the decoder generate comment edits in the form C_{edit}, we introduce experiments in which it directly generates C_{new}, with no intermediate edit sequence. For this, we use only the underlying architecture of the edit model(without features or reranking). The performance for various combinations of input code and target comment representations are shown in Table 5.4.

	Model	xM (%)	METEOR	BLEU-4	SARI	GLEU
Models	Edit	17.7	42.2	48.2	**46.4**	45.1
	- feats.	8.9†	36.2	44.3	40.5	39.9*
Reranked models	Edit	**18.4**	**44.7**	**50.7**	45.5	**46.1**
	- feats.	8.9†	38.4	46.7	36.9	40.3*

Table 5.5: Exact match, METEOR, BLEU-4, SARI, and GLEU scores of ablated models. Scores for which the difference in performance is *not* statistically significant ($p < 0.05$) are indicated with matching symbols.

By comparing performance across combinations consisting of the same input code representation and varying target comment representations, the importance of training the decoder to generate a sequence of edit actions rather than the full updated comment is very evident. Furthermore, comparing across varying code representations under the C_{edit} target comment representation, it is clear that explicitly encoding the code changes, as M_{edit}, leads to significant improvements across most metrics. We further ablate the explicit features. As shown in Table 5.5, these features improve performance by wide margins, across all metrics.

5.9 Additional Details

5.9.1 Model Parameters

Model parameters are identical across the edit model and generation model, tuned on validation data. Encoders have hidden dimension 64, the decoder has hidden dimension 128, and the dimension for code and comment embeddings is 64. The embeddings used in the edit model are initialized using the pre-trained embedding vectors from the generation model. We use a dropout rate of 0.6, a batch size of 100, an initial learning rate of 0.001, and Adam optimizer. Models

	Train	Valid	Test
Total actions	8,350	1,038	1,046
Avg. # actions per example	1.44	1.46	1.42
`Replace`	51.9%	49.7%	50.1%
`ReplaceKeepBefore`	2.9%	2.6%	3.5%
`ReplaceKeepAfter`	0.7%	0.3%	0.4%
`InsertKeepBefore`	21.5%	24.1%	23.2%
`InsertKeepAfter`	4.2%	4.0%	3.3%
`Delete`	17.4%	18.0%	17.8%
`DeleteKeepBefore`	1.3%	0.7%	1.1%
`DeleteKeepAfter`	0.2%	0.5%	0.6%

Table 5.6: Total number of edit actions; average number of edit actions per example; percentage of total actions that is accounted by each edit action type.

are trained to minimize negative log likelihood, and we terminate training if the validation loss does not decrease for ten consecutive epochs. During inference, we use beam search with beam width=20.

5.9.2 Modified Comment Edit Lexicon

We first transform insertions and ambiguous deletions into a structure that resembles `Replace`, characterized by `InsertOld` and `InsertNew` spans for insertions and `DeleteOld` and `DeleteNew` spans for deletions. Next, we require the span of tokens attached to `ReplaceOld`, `InsertOld`, and `DeleteOld` to be unique across C_{old} so that we can uniquely identify the edit location. We enforce this by iteratively searching through unchanged tokens before and after the span, incorporating additional tokens into the span, until the span becomes unique. These added tokens are then included in both components of the action. For instance, if the last `A` is to be replaced with `C` in `ABA`, the `ReplaceOld` span would be `BA` and the `ReplaceNew` span would be `BC`. We also augment the edit types to differentiate

between the various scenarios that may arise from this search procedure.

`Replace` actions for which this procedure is performed deviate from the typical nature of `Replace` in which there is no overlap between the spans attached to `ReplaceOld` and `ReplaceNew`. This is because the tokens that are added to make the `ReplaceOld` span unique will appear in both spans. These tokens, which are effectively kept between C_{old} and C_{new}, could appear before or after the edit location. We differentiate between these scenarios by augmenting the edit lexicon with new edit types. In addition to `Replace`, we have `ReplaceKeepBefore` and `ReplaceKeepAfter` to signify that the action entails retaining some content before or after, respectively.

We include the same for the other types as well with `InsertKeepBefore`, `InsertKeepAfter`, `DeleteKeepBefore`, `DeleteKeepAfter`. Table 5.6 shows statistics on how often each of these edit actions are used. Note that we disregard basic `Insert` actions since it is always ambiguous where an insertion should occur without an anchor point. We provide details about individual actions below.

Replace

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

This action prescribes that the tokens attached to `ReplaceOld` are deleted and the tokens attached to `ReplaceNew` are inserted in their place. There is almost never overlap between the span of tokens attached to `ReplaceOld` and `ReplaceNew`. Example: if `B` is to be replaced with `C` in C_{old}=`AB` to produce C_{new}=`AC`, the corre-

sponding C_{edit} is:

```
<ReplaceOld>B
<ReplaceNew>C
<ReplaceEnd>
```

Note that the span attached to `ReplaceOld` must be unique across C_{old} for this edit type to be used.

`ReplaceKeepBefore`

```
<ReplaceOldKeepBefore>[old span]
<ReplaceNewKeepBefore>[new span]
<ReplaceEnd>
```

`Replace` is transformed into this structure if the span attached to `ReplaceOld` is not unique. For example, suppose the first `B` is to be replaced with `D` in C_{old}=`ABCB` to produce C_{new}=`ADCB`. If C_{edit} consists of a `ReplaceOld` span carrying just `B`, it is not obvious whether the first or last `B` should be replaced. To address this, we introduce a new edit type, `ReplaceKeepBefore`, which forms a unique span by searching before the edit location.

It prescribes that the tokens attached to `ReplaceOldKeepBefore` are deleted and the tokens attached to `ReplaceNewKeepBefore` are inserted in their place. Unlike `Replace`, there will be some overlap at the beginning of the spans attached to `ReplaceOldKeepBefore` and `ReplaceNewKeepBefore`. To represent edits C_{old}=`ABCB` to produce C_{new}=`ADCB`, C_{edit} is:

```
<ReplaceOldKeepBefore> AB
<ReplaceNewKeepBefore> AD
<ReplaceEnd>
```

The span attached to `ReplaceOldKeepBefore` is unique, making it clear that the first `B` is to be replaced with `D`. It also indicates that we are effectively keeping `A`, before the edit location.

`ReplaceKeepAfter`

```
<ReplaceOldKeepAfter>[old span]
<ReplaceNewKeepAfter>[new span]
<ReplaceEnd>
```

`Replace` is transformed into this structure if the span attached to `ReplaceOld` is not unique and `ReplaceKeepBefore` cannot be used because we are unable to find a unique sequence of unchanged tokens before the edit location. For example, suppose the first `B` is to be replaced with `D` in C_{old}=`ABCAB` to produce C_{new}=`ADCAB`. Searching before the edit location, we find only `AB`, which is not unique across C_{old}, and so it would still not be clear which `B` is to be edited. To address this, we introduce a new edit type, `ReplaceKeepAfter`, which forms a unique span by searching *after* the edit location.

It prescribes that the tokens attached to `ReplaceOldKeepAfter` are deleted and the tokens attached to `ReplaceNewKeepAfter` are inserted in their place. Unlike `Replace` and `ReplaceKeepBefore`, there will be some overlap at the end of the spans attached to `ReplaceOldKeepAfter` and `ReplaceNewKeepAfter`. Therefore, to represent editing C_{old}=`ABCAB` to produce C_{new}=`ADCAB`, C_{edit} is:

```
<ReplaceOldKeepAfter> BC
<ReplaceNewKeepAfter> DC
<ReplaceEnd>
```

The span attached to `ReplaceOldKeepAfter` is unique, making it clear that the first `B` is to be replaced with `D`. It also indicates that we are effectively keeping `C`, which appears after the edit location.

`InsertKeepBefore`

```
<InsertOldKeepBefore>[old span]
<InsertNewKeepBefore>[new span]
<InsertEnd>
```

In this representation, the span of tokens attached to `InsertOldKeepBefore` must be unique and serve as the anchor point for where the new tokens should be inserted. We do this by searching before the edit location. The structure is identical to that of `ReplaceKeepBefore` in that the tokens attached to `InsertOldKeepBefore` are replaced with the tokens in `InsertNewKeepBefore` and that there is some overlap at the beginning of the two spans. As an example, suppose `C` is to be inserted at the end of C_{old}=`AB` to form C_{new}=`ABC`. Then the corresponding C_{edit} is as follows:

```
<InsertKeepBefore> B
<InsertNewKeepBefore> BC
<InserteEnd>
```

This states that we are effectively inserting `C` and keeping `B`, which appears before the edit location.

InsertKeepAfter

```
<InsertOldKeepAfter>[old span]
<InsertNewKeepAfter>[new span]
<InsertEnd>
```

We rely on this when we are unable to use `InsertKeepBefore` because we cannot find a unique span of tokens to identify the anchor point, by searching before the edit location. For instance, suppose `C` is to be inserted at the beginning of C_{old}=`AB` to form C_{new}=`CAB`. There are no tokens that appear before the insert point, so we instead choose to search *after*. The structure is identical to that of `ReplaceKeepAfter` in that the tokens attached to `InsertOldKeepAfter` are replaced with the tokens in `InsertNewKeepAfter` and that there is some overlap at the end of the two spans. The corresponding C_{edit} from our example is as follows:

```
<InsertKeepAfter> A
<InsertNewKeepAfter> CA
<InserteEnd>
```

This states that we are effectively inserting `C` and keeping `A`, which appears after the edit location.

5.9.3 Deletions

Delete

```
<Delete>[old span]<DeleteEnd>
```

It prescribes that the tokens that appear in the `Delete` span are removed from C_{old}. Example: if `B` is to be deleted from C_{old}=`AB` to produce C_{new}=`A`, the corresponding C_{edit} is:

```
<Delete>B<DeleteEnd>
```

Note that the `Delete` span must be unique across C_{old} for this edit type to be used.

`DeleteKeepBefore`

```
<DeleteOldKeepBefore>[old span]
<DeleteNewKeepBefore>[new span]
<DeleteEnd>
```

`Delete` is transformed into this structure if the `Delete` span is not unique. For example, suppose the first `B` is to be deleted from C_{old}=`ABCB` to produce C_{new}=`ACB`. From just C_{edit}=`<Delete>B<DeleteEnd>`, it is unclear which `B` is to be deleted. To address this, we introduce a new edit type, `DeleteKeepBefore`, which forms a unique span by searching before the edit location. The structure is identical to that of `ReplaceKeepBefore` in that the tokens attached to `DeleteOldKeepBefore` are replaced with the tokens in `DeleteNewKeepBefore` and that there is some overlap at the beginning of the two spans. For the example under consideration, the corresponding C_{edit} is given below:

```
<DeleteOldKeepBefore> AB
<DeleteNewKeepBefore> A
<DeleteEnd>
```

The span attached to `DeleteOldKeepBefore` is unique, making it clear that the

first B is to be deleted. It also indicates that we are effectively keeping A, which appears before the edit location.

DeleteKeepAfter

```
<DeleteOldKeepAfter>[old span]
<DeleteNewKeepAfter>[new span]
<DeleteEnd>
```

Delete is transformed into this structure if the Delete span is not unique and DeleteKeepBefore cannot be used because we are unable to find a unique sequence of unchanged tokens before the edit location. For example, suppose the first B is to be deleted from C_{old}=ABCAB to produce C_{new}=ACAB. Searching before the edit location, we find only AB, which is not unique across C_{old}, and so it would still not be clear which B is to be deleted. To address this, we introduce a new edit type, DeleteKeepAfter, which forms a unique span by searching *after* the edit location. The structure is identical to that of ReplaceKeepAfter in that the tokens attached to DeleteOldKeepAfter are replaced with the tokens in DeleteNewKeepAfter and that there is some overlap at the end of the two spans. For the example under consideration, C_{edit} is as follows:

```
<DeleteOldKeepAfter> BC
<DeleteNewKeepAfter> C
<DeleteEnd>
```

The span attached to DeleteOldKeepAfter is unique, making it clear that the first B is to be deleted. It also indicates that we are effectively keeping C, which

appears after the edit location.

5.9.4 Sample Output

In Table 5.7, we show predictions for various examples in the test set.

5.10 Summary

In this work, we have addressed the novel task of automatically updating an existing programming comment based on changes to the related code. We designed a new approach for this task which aims to correlate cross-modal edits in order to generate a sequence of edit actions specifying how the comment should be updated. We find that our model outperforms multiple rule-based baselines and comment generation models, with respect to several automatic metrics and human evaluation.

Examples

Project: ariejan-slick2d

```
 public float getX() {
-     return center[NUM];
 }
```

Old: @return the x location of the center of this circle

```
 public float getX() {
+    if (left == null) {
+        calculateLeft();
+    }
+    return left.floatValue();
 }
```

Base: @return the x location of the center of this circle or null if null

Gen: @return the x of the angle in this vector

Edit: @return the x location of the left of this circle

Gold: @return the x location of the left side of this shape .

Project: jackyglony-objectiveeclipse

```
 private IProject getProject() {
-     return managedTarget.getOwner().getProject();
 }
```

Old: @return the iproject associated with the target

```
 private IProject getProject() {
+     return (IProject) managedProject.getOwner();
 }
```

Base: @return the iproject associated with the target

Gen: @return the iproject

Edit: @return the iproject associated with the project

Gold: @return the iproject associated with the managed project

Project: rajawali-rajawali

```
 public double getRotX() {
-      return mOrientation.getRotationX();
 }
```

Old: @return double the roll euler angle .

```
 public double getRotX() {
+     return Math.toDegrees(mOrientation.getRotationX());
 }
```

Base: @return double the roll euler angle .

Gen: @return the rot x .

Edit: @return parsed double the roll euler angle .

Gold: @return double the roll euler angle in degrees .

Project: Qihoo360-RePlugin

```
-public static <T extends Collection<?>> T validIndex(final T collection,
                                                      final int index) {
-   return validIndex(collection, index,
-       DEFAULT_VALID_INDEX_COLLECTION_EX_MESSAGE, Integer.valueOf(index));
 }
```

Old: @return the validated collection (never null for method chaining)

```
+public static <T extends CharSequence> T validIndex(final T chars,
                                                     final int index) {
+   return validIndex(chars, index,
+       DEFAULT_VALID_INDEX_CHAR_SEQUENCE_EX_MESSAGE, Integer.valueOf(index));
 }
```

Base: @return the validated collection (never null for method chaining)

Gen: @return the index

Edit: @return the validated char sequence (never null for method chaining)

Gold: @return the validated character sequence (never null for method chaining)

Project: orfjackal-hourparser

```
 public Date getStart() {
     if (records.size() == NUM) {
-        return null;
     } else {
         Date first = records.get(NUM).getDate();
         for (Entry e : records) {
             if (e.getDate().before(first)) {
                 first = e.getDate();
             }
         }
         return first;
     }
 }
```

Old: @return the time of the first record or null if there are no records

```
 public Date getStart() {
     if (records.size() == NUM) {
+        return new Date();
     } else {
         Date first = records.get(NUM).getDate();
         for (Entry e : records) {
             if (e.getDate().before(first)) {
                 first = e.getDate();
             }
         }
         return first;
     }
 }
```

Base: @return the time of the first record or null if there are no records

Gen: @return the date , or null if not available

Edit: @return the time of the first record or date if there are no records

Gold: @return the time of the first record , or the current time if there are no records

Table 5.7: Examples from open-source software projects. For each example, we show the diff between the two versions of the method (left: old version, right: new version, diff lines are highlighted), the existing `@return` comment prior to being updated (left), and predictions made by the *return type substitution w/ null handling* baseline, reranked generation model, and reranked edit model, and the gold updated comment (right, from top to bottom).

Chapter 6

Combined Detection and 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. This chapter is based on the work originally presented in Panthaplackel et al. (2021).

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 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

	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.

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.

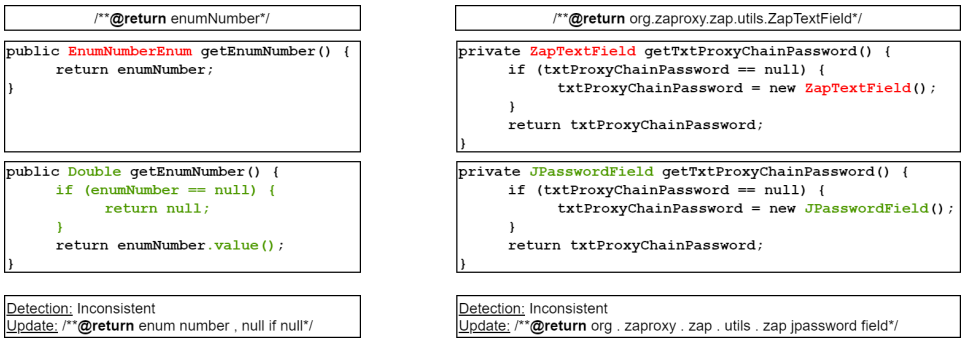
	Update Metrics				
	xMatch	**METEOR**	**BLEU-4**	**SARI**	**GLEU**
Never Update	50.0	67.7	71.6	25.1	68.3
Update model (Chapter 5)	21.5	56.2	64.7	37.6*	63.4
Update w/ implicit detection	56.1*	71.3	73.4*	30.2	71.4
Pretrained update + detection					
SEQ(C_{old}, M_{edit}) + features	**57.3**§	**72.6***	**73.9**†	37.8§	**73.2**§
GRAPH(C_{old}, T_{edit}) + features	55.2	71.8	73.5*	38.0†‖	72.8*
HYBRID(C_{old}, M_{edit}, T_{edit}) + features	**57.3**§	**72.6***	**73.9**†	37.6*	**73.2**†§
Jointly trained update + detection					
SEQ(C_{old}, M_{edit}) + features	56.5*†	72.2†§	73.5*	37.9†‖	72.9*
GRAPH(C_{old}, T_{edit}) + features	56.2*	72.0§	73.6*	37.8†§	73.0*†
HYBRID(C_{old}, M_{edit}, T_{edit}) + features	56.8†	72.4*†	73.8	**38.1**‖	73.1§

Table 6.3: Comparing performance on update between combined systems on the full test set. 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)	53.1	91.8	67.2	55.3
Update w/ implicit detection	**98.5**	18.2	30.8	59.0
Pretrained update + detection				
SEQ(C_{old}, M_{edit}) + features	88.4†	73.2	80.0†	81.8*†
GRAPH(C_{old}, T_{edit}) + features	83.8	**78.3**	80.9*	81.5†
HYBRID(C_{old}, M_{edit}, T_{edit}) + features	88.6†	72.4	79.6†	81.5†
Jointly trained update + detection				
SEQ(C_{old}, M_{edit}) + features	85.7*	76.7*	80.9*	81.9*†
GRAPH(C_{old}, T_{edit}) + features	85.9*	76.7*	**81.0***	82.0*†
HYBRID(C_{old}, M_{edit}, T_{edit}) + features	86.7	75.7	80.9*	**82.1***

Table 6.4: 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.

In Tables 6.3 and 6.4, we show results of combined detection+update systems on the full test set. The results are analogous to those corresponding to the the cleaned test set. While the differences for the update metrics are less pronounced, the pretrained and jointly trained approaches can again outperform *Update w/ implicit detection* as well as the two reference points: Never Update and Update Model (Chapter 5). The drastic differences in performance with respect to the detection metrics further demonstrate the importance of explicit inconsistency detection in a



(a) Example from OpenAPI Generator (b) Example from OWASP ZAP

Figure 6.1: Examples in which inconsistencies emerged as a result of developers failing to update comments upon code changes. Predictions of the combined, pre-trained detection+update approach are shown.

combined detection+update system. In line with our observations from the cleaned test set, we find the performances of the pretrained and jointly trained systems to be very close.

6.3 Qualitative Analysis

Recall that in our data collection procedure, we assign the negative (i.e., consistent) label to examples in which the developer did not update a comment following code changes. Based on our inspection of a sample of the full, unannotated test set, we find examples that are mislabeled as negative, and our model can correctly identify some of these cases. For instance, in the example shown in Figure 6.1a, the developer failed to amend the comment to indicate that the method no longer returns `enumNumber` but rather its value or `null` if it is not set. Similarly, in Figure 6.1b, the developer failed to update `ZapTextField` to `JPasswordField` in the comment when the return type of the method was modified. The inconsis-

tency in the OWASP ZAP project was fixed after we reported the issue and the inconsistency in the OpenAPI Generator project continues to persist today. [1]

6.4 Summary

We explored various strategies for combining our approach for inconsistency detection (Chapter 4) with our approach for comment update (Chapter 5). We demonstrate that these approaches can be integrated to build a comprehensive comment maintenance system that can detect and resolve inconsistent comments.

[1] We have reported them as issues in their respective projects.

Chapter 7

Describing Solutions for Bug Reports Based on Developer Discussions

In Chapters 3-6, the underlying goal was to *support software evolution*, for which we focused on natural language comments. We developed techniques for detecting and updating comments immediately *after* code changes to uphold software quality once these changes are merged into the code base. In Chapters 7-8, we shift to our second goal which aims to *drive software evolution*. For this, we use a different form of natural language, namely dialogue in bug report discussions[1], to 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, in this chapter, 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. This chapter is based on work presented in Panthaplackel et al. (2022b).

[1] We provide an overview of bug report discussions in Section 2.4.

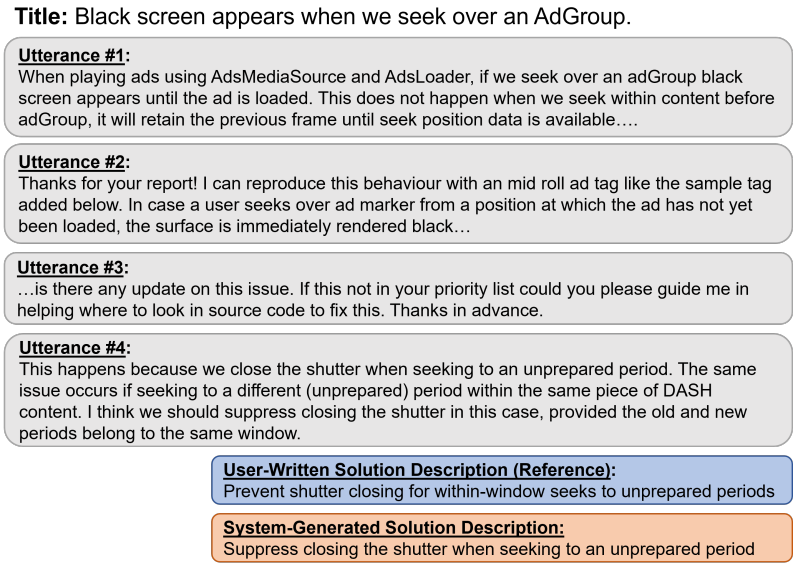


Figure 7.1: Bug report discussion from ExoPlayer[2] with user-written and system-generated solution descriptions.

7.1 Problem Setting

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. Other participants join the discussion at later time steps 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). We propose 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 new corpus which has been released publicly.[3] Following prior work on other tasks (Kavaler et al., 2017; Panichella et al., 2021), we mine issue reports corresponding to open-source Java projects from GitHub Issues. 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 heuristics 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 it 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 resulted in code (or documentation) fixes (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). In practice, commit messages and pull request titles are written after code changes. However, like contemporary work (Chakraborty and Ray, 2021), we treat them as a proxy for solution descriptions to drive bug-resolving code changes.

[3] `https://github.com/panthap2/describing-bug-report-solutions`

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 issues with multiple commit messages/PR titles, so there is at most one example per issue. This is because the reason for needing multiple sets of changes is not clear (e.g., the solution could be implemented in parts or the first solution may have been incorrect and it is later corrected).[4]

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 2.6). First, we apply simple heuristics to reduce noise, which we discuss in more detail in Section 7.8.2. From this, we obtain the examples that are primarily used for training and evaluation in this work, which we refer to as the *full dataset*. Next, we identify three sources of noise that are more difficult to control with simple heuristics and use techniques described below to quantify them and build a *filtered subset* of the full dataset that is less noisy. This subset is used for

[4]Since such examples could be useful for future work, they are available in the data we released.

more detailed analysis of the models that are discussed in the paper, and we find that training on this subset leads to improved performance (§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 the training data).

Uninformative descriptions: Instead of describing the solution, the commit message or pull request title sometimes essentially re-states 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. For instance, in the discussion in the footnote[5], 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 such examples with an

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

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 (i.e., there is no relevant sentence that is extracted from the context). After applying all three filters, we have 5,856 examples.

7.2.3 Preprocessing

We *subtokenize* (Section 3.3) the title, utterances, and description. We retain inlined code (on average 5.7 tokens/utterance); however, we remove code blocks and embedded code snippets (with markdown tags), 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). However, markdown tags are not always used to identify code (Tabassum et al., 2020), and consequently, we observe some instances of larger code blocks within utterances that cannot be easily removed. We do not use source code files within a project's repository and leave it to future work to incorporate large bodies of code. We discard URLs and mentions of GitHub usernames from utterances. From the description, we remove references to issue 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

	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.

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

We benchmark various models for generating 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 $\sim$70 tokens while the average description length is only $\sim$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,

		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.

rather than an *extractive* one which generates a description by merely copying over utterances or sentences within the discussion.[6] Furthermore, success 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 re-

[6] We observe very low performance with extractive approaches, as shown in Section 7.8.3.

sponse 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 finetuning, PLBART achieves state-of-the-art performance on many program and language understanding tasks like code summarization/generation. We finetune PLBART on our training set and evaluate its

	Model	BLEU-4	METEOR	ROUGE-L
Full	Copy Title	14.4$^{\|}$	13.1	24.4§
	SEQ2SEQ + Ptr	12.6	9.8	25.0‡
	Hier SEQ2SEQ + Ptr	12.4	9.6	24.1§
	PLBART	**16.6**	**14.5**	**28.3**
	PLBART (F)	14.2$^{\|}$	12.3	25.1‡
Filtered	Copy Title	10.0*†	8.3	16.6
	SEQ2SEQ + Ptr	10.2*	7.5	20.1
	Hier SEQ2SEQ + Ptr	9.9†	7.4	19.6
	PLBART	**12.3**‡	9.9	21.1
	PLBART (F)	**12.3**‡	**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 ($p < 0.05$) are indicated with matching symbols.

ability to comprehend bug report discussions and generate descriptions of solutions.[7] 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 finetuning data. So we finetune on only the filtered subset of the training set (Section 7.2.2), to investigate whether finetuning on this "less noisy" sample can lead to improved performance.

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

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

on large amounts of data[8]. 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 ratio-

[8]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.

nale for their selection. Before starting the annotation task, users must watch a training video in which we walk through seven 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 annotate 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 to the projects they were asked to review, 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;

Model	Full	Filtered
Copy Title	8.1	6.0
SEQ2SEQ + Ptr	1.3*	1.2†
Hier SEQ2SEQ + Ptr	1.3*	1.2†
PLBART	11.9	10.5
PLBART (F)	**33.1**‡	**39.5**
All Poor	20.0	22.1
Insufficient Context	31.9‡	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 ($p < 0.05$) are indicated with matching symbols.

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 finetuning 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.

7.6 Analysis

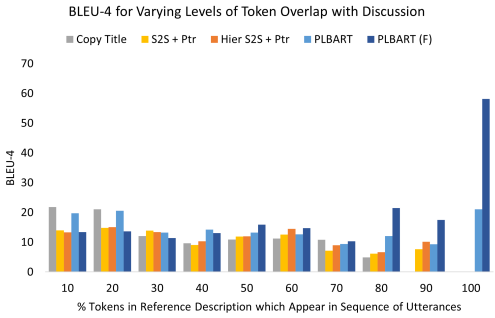
Of the 160 annotated examples in Table 7.4, users found 51 to have insufficient context about the solution. We consider the remaining 109 as the *context-sufficient subset (CS)*, which we released for future research. We present automated metrics for this subset in Table 7.5. Results are analogous to the full test set, ex-

Model	BLEU-4	METEOR	ROUGE
Copy Title	12.6	12.2¶	22.1
SEQ2SEQ + Ptr	11.6	8.9	23.1
Hier SEQ2SEQ + Ptr	12.0	9.0	22.9
PLBART	**14.6**	**13.2**	**26.0**
PLBART (F)	14.2	12.3¶	25.1

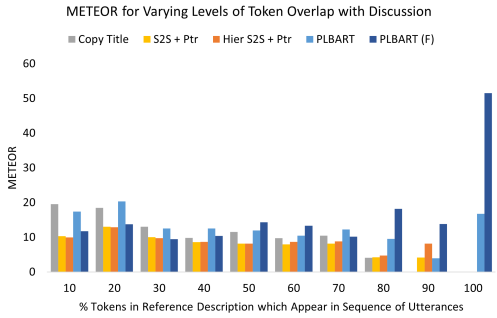
Table 7.5: Automated metrics for generation on CS subset. Differences that are *not* statistically significant are indicated with matching symbols.

cept that the numbers are generally lower for all models other than for PLBART (F), which achieves consistent performance. PLBART (F) slightly underperforms PLBART on automated metrics overall. However, this is because these metrics are computed against the single reference description, which could diverge from how the solution is formulated in the discussion since the developer could have written an uninformative/generic description. To do more fine-grained analysis, in Figure 7.2, we plot automated metrics for varying percentages of token overlap between the reference description and $U_1...U_{t_g}$ (excluding tokens already present in the title which have been used to state the problem). Higher overlap suggests that the reference description draws more content from within the discussion. For higher percentages, PLBART (F) generally achieves higher scores against the reference than PLBART and all other models, indicating that this model is better at gathering information from within the discussion.

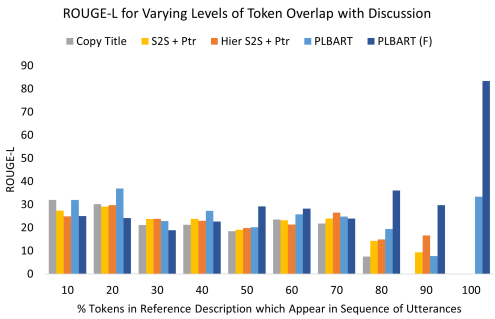
To analyze how models exploit the provided context, we measure the percent of n-grams in the prediction which overlap with the title and $U_1...U_{t_g}$ (excluding n-grams already in the title) in Table 7.6. PLBART (F)'s predictions tend to have less n-gram overlap with the title and more overlap with the utterances. This suggests that this model predicts fewer uninformative descriptions which merely re-state the problem mentioned in the title and instead focuses on content from the utterances.



(a) BLEU



(b) METEOR



(c) ROUGE

Figure 7.2: Metrics for CS subset, with buckets corresponding to the % of tokens in reference description which also appear in $U_1 ... U_{t_g}$ (disregarding title tokens). Bucket 10 corresponds to [0, 10)%, 20 corresponds to [10, 20)%, etc.

In Table 7.7, 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, we find that many times, PLBART (F) generates output that is informative towards bug resolution. While this demonstrates that finetuning this 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 clearly room for improvement.

We manually inspected PLBART (F)'s outputs and associated user ratio-

		Title ↓				$U_1 ... U_{t_g}$ only ↑			
	Model	**1**	**2**	**3**	**4**	**1**	**2**	**3**	**4**
Full	Copy Title	100.0	100.0	100.0	100.0	0.0	0.0	0.0	0.0
	SEQ2SEQ + Ptr	65.6	34.4	39.3	46.5	28.6	24.9	27.0	25.0
	Hier SEQ2SEQ + Ptr	60.2	33.9	41.1	50.4	37.4	27.9	28.3	29.2
	PLBART	79.3	75.0	72.5	71.7	30.7	34.8	34.6	**39.9**
	PLBART (F)	43.2	37.4	38.3	43.1	**47.1**	**38.1**	**35.6**	37.2
	Reference	**35.1**	**30.9**	**33.5**	**37.7**	34.5	22.2	22.2	25.3
Filtered	Copy Title	100.0	100.0	100.0	100.0	0.0	0.0	0.0	0.0
	SEQ2SEQ + Ptr	64.5	33.8	39.1	38.3	29.4	25.3	23.8	0.0
	Hier SEQ2SEQ + Ptr	58.4	33.3	39.3	45.7	40.4	28.4	30.0	29.2
	PLBART	76.9	73.4	71.1	70.4	34.0	37.0	36.3	**41.2**
	PLBART (F)	38.4	33.9	35.2	40.7	**51.0**	**40.0**	**36.6**	38.1
	Reference	**23.7**	**18.6**	**18.4**	**16.3**	40.1	22.8	21.4	23.0
CS	Copy Title	100.0	100.0	100.0	100.0	0.0	0.0	0.0	0.0
	SEQ2SEQ + Ptr	64.8	37.1	38.5	22.5	31.6	25.3	33.1	25.0
	Hier SEQ2SEQ + Ptr	60.3	34.2	37.9	28.3	38.7	26.1	29.2	0.0
	PLBART	80.8	77.7	72.8	70.3	31.0	41.4	37.0	**50.0**
	PLBART (F)	36.9	28.4	30.8	34.1	**52.8**	**42.3**	**39.4**	45.0
	Reference	**32.7**	**22.2**	**26.2**	**35.6**	38.8	25.4	23.1	27.1

Table 7.6: Percent of unigrams, bigrams, trigrams and 4-grams in the prediction (or reference) which appear in the title and in $U_1..U_{t_g}$ only (excluding the title). Lower is better for the title and higher is better for $U_1..U_{t_g}$ only.

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 7.7: Model outputs for the example shown in Figure 7.1.

nales. We observe that the model tends to perform better when the solution is clearly stated in 1-3 consecutive sentences (Table 7.8 (1) and (2)). When more complex synthesis is needed, it sometimes stitches together tokens from the input incorrectly (Table 7.8 (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 7.8 (4)). This demonstrates that it still struggles to disentangle content relevant to the solution

	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 7.8: Output of PLBART (F) for a sample of examples in the test set.

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 7.8 (5)). Finally, we find problems with repetition and fluency (Table 7.8 (1)), as commonly seen in the outputs of neural models (Holtzman et al., 2020).

7.7 Supporting Real-Time Generation

Generating an informative description requires sufficient context about the solution being available in the discussion. In a real-time setting, this context is likely not immediately available but rather emerges as the discussion progresses, and we must wait until it becomes available to generate a solution description. However, the time step at which it becomes available (t_g) is not known beforehand, so we must instead predict it (t_p) in order to perform generation *during* ongoing discus-

sions. For this, we consider classifying whether sufficient context is available upon each new utterance. In Figure 7.1, the solution is formulated in U_4, so the correct behavior is to predict the negative label at $t = 1, 2, 3$ and the positive label at $t = 4$. Once the positive label is predicted at t_p[9], the description is generated, conditioned on the title and $U_1 ... U_{t_p}$. We develop two systems for integrating classification with a generation model: *pipelined* and *joint trained*.

7.7.1 Pipelined System

We design an independent classifier built on PLBART's encoder. When a new utterance U_t is made in the discussion, we encode the context so far (the title and all utterances up to and including U_t). We take the final hidden state, e_t, as the context representation at t, which we feed e_t through a 3-layer classification head and apply softmax to classify whether or not sufficient context is available. We train to minimize cross entropy loss. At test time, we use the already trained PLBART (F) model to generate a solution description with context available at t_p.

7.7.2 Joint System

We initialize an encoder-decoder model from PLBART with an additional classification head (§7.7.1). The encoder is shared among the two tasks. When classifying whether sufficient context about the solution is available, there is likely specific solution-related content that contributes to predicting the positive label. So, classification may enhance encoder representations, improving content selection for generating solution descriptions.

[9]Classification is not performed for $t > t_p$.

Furthermore, having sufficient context correlates with whether it can be used to generate an *informative* description. So, the informativeness of a description that can be generated with the available context can provide signal for classifying whether that context is sufficient. Additionally, if sufficient context was not previously available at $t-1$ but becomes available at t, we expect an improvement in the informativeness of the descriptions generated at the two time steps. We represent these descriptions with the final decoder states at the two time steps, d_{t-1} and d_t. We concatenate e_t, d_{t-1}, and d_t to form the input into the classification head.[10] For training loss, we sum the generation and classification losses across time steps $t_1 ... t_g$. Sufficient context for generation may not be available at $t < t_g$, so we mask generation loss for earlier time steps.

7.7.3 Evaluation

We train on filtered data since we found this to improve performance. At test time, a system can generate a solution description at $t_p \leq t_g$, or it can fail to predict the positive label before or at t_g. After a commit/PR for fixing the bug is made at t_g, the state of the discussion changes, with possible mentions of the solution that is implemented. Since using this as context to generate a solution description can be considered "cheating," we do not make predictions for time steps after t_g. We treat this as the system *refraining* from generating after not finding sufficient context.

The pipelined and joint systems refrained from generating 33.3-35.4% and

[10] In initial experiments, we tried explicitly quantifying the extent of improvement in informativeness between the timesteps by feeding in the difference vector, $d_t - d_{t-1}$, instead of two separate vectors. However, we found that this did not perform as well, and it was better to let the model learn to reason about the differences.

		$\mathbf{t_p \leq t_g}$	$\mathbf{t_g - t_p}$	**BLEU**	**METEOR**	**ROUGE**
Pipelined	Full	$@t_p$	1.69	$14.3^{\ddagger}$	$12.4^{\S}$	$25.1^{\P}$
		$@t_g$	-	**14.4**‡	**12.5**§	**25.3**¶
	Filtered	$@t_p$	1.85	12.5^{*}	10.1	21.7
		$@t_g$	-	**12.6***	**10.5**	**22.3**
Joint	Full	$@t_p$	1.81	13.1	11.4	$22.4^{\dagger}$
		$@t_g$	-	**13.2**	**11.7**	**22.5**†
	Filtered	$@t_p$	1.97	11.7	9.5	19.3
		$@t_g$	-	**11.9**	**9.9**	**19.7**

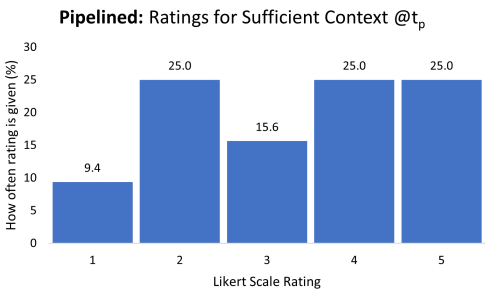
Table 7.9: Automated metrics for combined systems when $t_p \leq t_g$. We compare the generated description $@t_p$ with that if the system had generated $@t_g$. Differences that are *not* statistically significant are indicated with matching superscripts.

		$\mathbf{t_g - t_p}$	**BLEU**	**METEOR**	**ROUGE**
Full	Pipelined	2.09	**14.4**	**12.4**	**24.8**
	Joint	**1.86**	12.9	11.3	22.3
Filtered	Pipelined	2.16	**12.4**	**10.0**	**21.0**
	Joint	**2.03**	11.4	9.2	18.7

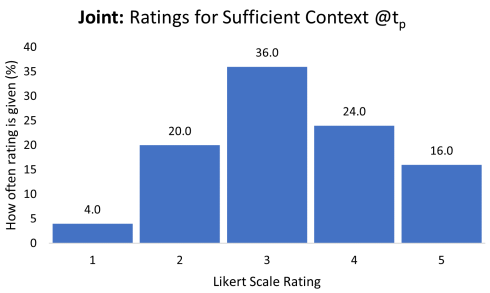
Table 7.10: Performance at t_p on examples for which both systems predicted $t_p \leq t_g$ (614 of full and 304 of filtered test sets). All differences are statistically significant.

36.4-39.8% of the time respectively. We present automated metrics for the remaining cases in Table 7.9. We find that $t_g - t_p$ is between 1.69 and 1.85 for the pipelined system and between 1.81 and 1.97 for the joint system. While a system should wait until sufficient context is available, sometimes, the last couple utterances before the implementation do not add context about the solution but are personal exchanges (e.g., "Thanks", "I'll open a PR"). So, generating slightly before t_g is acceptable in some cases. Moreover, despite generating early in some cases, the generated output $@t_p$ achieves comparable performance to that $@t_g$, with respect to the generation metrics (BLEU, METEOR, and ROUGE).

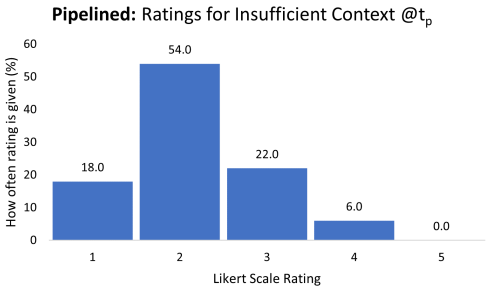
Note that the numbers are not directly comparable across the two systems since the exact subset of examples for which $t_p \leq t_g$ varies between the two. In



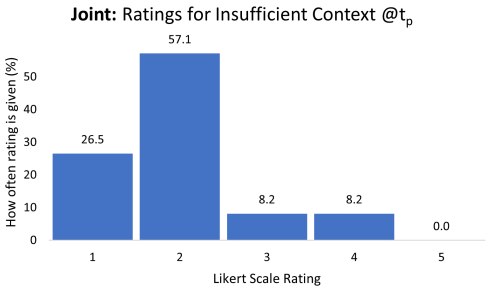
(a) Pipelined: Sufficient Context @t_p



(b) Joint: Sufficient Context @t_p



(c) Pipelined: Insufficient Context @t_p



(d) Joint: Insufficient Context @t_p

Figure 7.3: The distribution of Likert scale ratings for the pipelined and jointly trained systems, presented separately for the cases in which there is sufficient context @t_p and there is insufficcient context @t_p. Note that numbers cannot be directly compared across systems, as the exact examples for which generation is performed varies.

Table 7.10, we present results for the subset of examples for which both systems predict $t_p \leq t_g$. The joint system achieves lower average error ($t_g - t_p$) for classification while the pipelined system performs better on generation metrics.

We also do human evaluation, for which we recruited 6 graduate students with 3+ years of Java experience. Each user evaluated outputs of the two systems for 20 random examples from the filtered test set. Users are given the same information as Section 7.5. If the system refrained from generating, we ask them if there is sufficient context about the solution at any time step $t \leq t_g$. Otherwise,

we show them the generated description and ask if there is sufficient context about the solution at t_p and also to rate the informativeness of the description on a Likert scale: 1: incomprehensible, completely incorrect, irrelevant; 2: generic, rephrasing problem; 3: includes some useful information but does not capture the solution; 4: partially captures solution; 5: completely captures solution.

In the cases that the system generated a description, users found there to be sufficient context at t_p 39.0% and 33.8% of the time for the pipelined and joint systems respectively, with average informativeness being 3.3 for both (distributions of ratings in Figures 7.3a and 7.3b). This suggests that when sufficient context is available, these systems generate descriptions which can be useful for bug resolution.

Because a real-time system must act at a given time step agnostic to future activity, classifying *when to generate* is challenging. It should defer generation to later time steps if the optimal context is not available. Generating too early can result in output that is generic and re-states the problem. For the cases in which the system generated a description *without* sufficient context at t_p, the average informativeness ratings were 2.2 (pipelined) and 2.0 (joint). Based on the distributions of ratings presented in Figures 7.3c and 7.3d, we see that model predictions are generally given low informativeness ratings for cases with insufficient context. However, deferring generation for too long by expecting more context to emerge later also poses a risk. After the solution has already been implemented, it is too late for a generated description to be useful towards resolving the bug. In the cases that the pipelined and joint systems refrained from generating, there was sufficient context

about the solution 34.2% and 37.0% of the time respectively.

Despite the pipelined and joint systems having nuanced differences, we find them to perform similarly. Through our evaluation of these systems, we demonstrate room for improvement, particularly for the classification component in determining the optimal time step for generation. We leave it to future work to develop more intricate end systems.

7.8 Additional Details

7.8.1 Model Parameters

All neural models were implemented using PyTorch. For SEQ2SEQ + Ptr and Hier SEQ2SEQ + Ptr, we use a batch size of 8, an initial learning rate of 3e-05, and a dropout rate of 0.2. Our transformer models have 4 encoder and decoder layers, 4 heads in multi-head attention, a hidden size of 64, and feedforward hidden size 256. We use Adam as the optimizer and have a learning rate scheduler with gamma 0.95 which decays after an epoch if the validation loss has not improved. We use early stopping with patience 5 during training.

For classification, the classification head consists of a linear layer (dimension 768), followed by a tanh non-linear layer, and a final linear projection layer (dimension 2). When computing cross entropy loss for classification, we weight the positive and negative labels using the inverse of the class proportion to handle class imbalance (1.70 and 0.71 respectively). For the joint model, loss for a given example is computed as follows, with $\lambda_1 = 0.8, \lambda_2 = 0.2$ (tuned on validation

data).

$$L = \lambda_1 L_{gen}(t_g) + \lambda_2 \sum_{t=1}^{t=t_g} L_{class}(t)$$

7.8.2 More Data Details

We focus on closed bug reports from the top 1,000 Java projects (in terms of number of stars), as a way of identifying well-maintained projects (Section 3.2). We require there to be at least two distinct "actors" in the discussion, in which the actor can either be a developer who makes an utterance in the discussion or an actor who implements the solution through a commit or pull request. We discard examples in which the reference description is identical to the title (disregarding stopwords), as these are cases in which either the reference description only states the problem and is uninformative or the title already puts forth a solution (in which case a generated description would not be useful). We remove examples with commits or pull requests which simultaneously address multiple bug reports.

We mined 141,389 issues (from 770 of the top 1,000 projects). After applying heuristics, we get 35,010 (from 525 projects), which will be released. Of these, 16,899 pertain to bugs and 18,111 pertain to non-bugs. From the 16,899 bug-related issues, we focus on the 12,328 issues with a single commit message/PR title. We explain our reasoning for discarding examples linked to multiple commits and/or pull requests in Section 7.2.1. However, such examples (which are available in the data we release) can be useful for supporting generating descriptions at multiple time steps in future work.

From an example's description, we remove references to issue and pull re-

quest numbers, as they do not contribute to the meaning and are instead used as identifiers for organizational purposes.

7.8.3 Additional Generation Baselines

We considered additional baselines; however, since they were performing much lower than other approaches (on wide statistically significant margins), we chose to exclude them from the main results. We briefly describe these baselines below.

Extractive Baselines

Supervised Extractive: Using a greedy approach for obtaining noisy extractive summaries (Nallapati et al., 2017), we train a supervised extractive summarization model, similar to (Liu and Lapata, 2019).

LexRank: We use LexRank (Erkan and Radev, 2004), an unsupervised graph-based extractive summarization approach. We extract a single sentence, with threshold 0.1.

$\mathbf{U_1}$ **(Lead 1)**: This entails simply taking the first sentence of the first utterance, intended to simulate the Lead-1 baseline that is commonly used in summarization.

$\mathbf{U_1}$ **(Lead 3)**: This entails simply taking the first 3 sentences of the first utterance, intended to simulate the Lead-3 baseline that is commonly used in summarization.

$\mathbf{U_{t_g}}$: Since some part of the solution is often mentioned within U_{t_g}, we copy this utterance.

$\mathbf{U_{t_g}}$ **(Lead 1)**: Since the length of an utterance is quite different than that of a

description (Table 7.1), we extract only the lead sentence of U_{t_g}.

$\mathbf{U_{t_g}}$ **(Lead 3)**: For the reason stated above, we also apply the Lead-3 baseline to this utterance.

$\mathbf{U_{t_g}}$ **(Last sentence)**: Rather than extracting the lead sentence, we extract the last sentence of U_{t_g}.

$\mathbf{U_{t_g}}$ **(Last 3 sentences)**: Rather than extracting the lead 3 sentences, we try extracting the last 3 sentences of U_{t_g}.

Retrieval Baselines

Retrieval (Title-Title): Using TF-IDF, we compute cosine similarity between the test example's title and titles in the training set, to identify the closest training example, from which we take the description.

Retrieval (Title-Desc): Using TF-IDF, we compute cosine similarity between the test example's title and *descriptions* in the training set, to identify the closest training example, from which we take the description.

Project Retrieval (Title-Title): Using TF-IDF, we compute cosine similarity between the test example's title and titles *for the same project* in the training set, to identify the closest training example, from which we take the description.

Project Retrieval (Title-Desc): Using TF-IDF, we compute cosine similarity between the test example's title and descriptions for the same project in the training set, to identify the closest training example, from which we take the description.

	Model	BLEU	**METEOR**	**ROUGE**
Full	Supervised Extractive	0.5	0.5	0.8
	LexRank	2.2	1.9	2.5
	U_1 (Lead 1)	4.8	6.5	8.8
	U_1 (Lead 3)	3.1	8.0	8.7
	U_{t_g}	2.8	5.4	6.7
	U_{t_g} (Lead 1)	4.0	4.5	6.9
	U_{t_g} (Lead 3)	3.2	5.7	7.4
	U_{t_g} (Last sentence)	3.5	3.5	5.5
	U_{t_g} (Last 3 sentences)	3.2	5.1	6.8
	Retrieval (Title-Title)	6.9	4.5	10.7
	Retrieval (Title-Desc)	8.8	6.2	14.8
	Project Retrieval (Title-Title)	7.4	4.7	10.9
	Project Retrieval (Title-Desc)	9.1	6.3	14.1
	Copy Title	14.4	13.1	24.4
	SEQ2SEQ + Ptr	12.6	9.8	25.0
	Hier SEQ2SEQ + Ptr	12.4	9.6	24.1
	PLBART	**16.6**	**14.5**	**28.3**
	PLBART (F)	14.2	12.3	25.1
Filtered	Supervised Extractive	0.7	0.7	1.0
	LexRank	2.4	1.9	2.6
	U_1 (Lead 1)	5.0	6.2	8.6
	U_1 (Lead 3)	3.1	7.9	8.8
	U_{t_g}	2.9	6.0	7.3
	U_{t_g} (Lead 1)	4.4	4.8	7.6
	U_{t_g} (Lead 3)	3.4	6.3	8.1
	U_{t_g} (Last sentence)	3.5	4.0	5.9
	U_{t_g} (Last 3 sentences)	3.3	5.7	7.4
	Retrieval (Title-Title)	6.1	3.7	9.0
	Retrieval (Title-Desc)	7.0	4.5	11.4
	Project Retrieval (Title-Title)	6.6	4.2	9.2
	Project Retrieval (Title-Desc)	7.6	5.1	11.3
	Copy Title	10.0	8.3	16.6
	SEQ2SEQ + Ptr	10.2	7.5	20.1
	Hier SEQ2SEQ + Ptr	9.9	7.4	19.6
	PLBART	**12.3**	9.9	21.1
	PLBART (F)	12.3	**10.2**	**21.9**

Table 7.11: Comparing the main models with low-performing baselines for generating solution descriptions. Scores for Supervised Extractive are averaged across three trials.

Baseline Results

We present baseline results in Table 7.11. All of these baselines substantially underperform models presented in Table 7.3, especially the Supervised Extractive model. We believe this model performs so poorly due to noise in the supervision and because the extracted summaries are longer and structured differently than the reference descriptions in our dataset. Additionally, there are many examples in which the model does not select a single sentence from the input, resulting in the prediction being the empty string. LexRank also performs poorly in terms of automated metrics against the reference description. This unsupervised approach aims to identify a "centroid" sentence that summarizes the full input context and is not designed to specifically focus on solution-related context.

All baselines that extract a whole utterance or sentences from specific utterances perform poorly, demonstrating the need for content selection from the broader context and content synthesis rather than relying on simple heuristics to produce a description of the solution. We find that the retrieval baselines tend to achieve higher scores, as retrieved descriptions are from the same distribution as the reference descriptions. However, these numbers are still much lower than those in Table 7.3.

7.9 Classification Baselines

To benchmark performance on the classification task for determining when sufficient context is available for generating an informative description, we consider

		FIRST	SECOND	RAND (uni)	RAND (dist)	Pipelined	Joint
Full	(↑) $\mathbf{t_p} \leq \mathbf{t_g}$	100.0%	70.5%	76.0%	77.1%	66.7%	60.2%
	(↓) $\mathbf{t_g} - \mathbf{t_p}$	2.2	2.1	2.2	2.2	1.7	1.8
Filtered	(↑) $\mathbf{t_p} \leq \mathbf{t_g}$	100.0%	76.2%	79.4%	80.1%	64.6%	63.6%
	(↓) $\mathbf{t_g} - \mathbf{t_p}$	2.6	2.4	2.5	2.5	1.9	2.0

Table 7.12: Percent of time $t_p \leq t_g$ and for these particular cases, the mean absolute error between t_g and t_p.

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).

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 also include RAND (dist), which instead uses the probability distribution of labels at the example-level estimated from the filtered training set (pos = $\frac{1}{N}\sum_{n=1}^{N}\frac{1}{t_g}$=0.510, neg = 0.490). Results are averaged across 3 trials. We present results in Table 7.12.

7.10 Summary

We presented the novel task of generating concise natural language solution descriptions to guide developers in absorbing information relevant towards bug resolution from long discussions. We established benchmarks for this task using a dataset that we constructed with supervision derived from commit messages and pull request titles. Through automated and human evaluation, we demonstrated the

utility of these models and also highlight their shortcomings, to encourage more research in exploring ways to address these challenges. We also simulated a real-time setting through two approaches for combining a generation model with a classification component for determining when sufficient context for generating an informative description emerges in an ongoing discussion. We believe this lays the groundwork for future work on building a dialogue agent that participates in bug report discussions to foster efficient resolution.

Chapter 8

Using Bug Report Discussions to Guide Automated Bug Fixing

In Chapter 7, we proposed generating natural language solution descriptions based on bug report discussions. While this can provide a high-level overview of the solution, developers must still reason about how it should manifest as concrete code changes in order to actually fix the bug. Due to the extensive developer time and effort needed to fix bugs (Weiss et al., 2007), there is growing interest in automated bug fixing (Tufano et al., 2019; Chen et al., 2019b; Lutellier et al., 2020; Mashhadi and Hemmati, 2021; Allamanis et al., 2021; Chakraborty and Ray, 2021). While recent work (Chakraborty and Ray, 2021) showed that natural language context is useful in guiding bug-fixing models, their approach required prompting developers to provide this context, which was simulated through commit messages written *after* the bug-fixing code changes were made. We instead propose using bug report discussions, which are available *before* the task is performed and are also naturally occurring, avoiding the need for any additional information from developers. This chapter is based on work presented in Panthaplackel et al. (2022a).

```
void emptyImplicitTable(String table, int line) {
    sb.append("Invalid table definition due to
              empty implicit table name: ")
      .append(table)
      .append("\n");                                  Buggy

    sb.append("Invalid table definition due to
              empty implicit table name: ")
      .append(table);                                 Fixed
}
```

Oracle Commit Message: Removed trailing newlines from error messages. Fixes https://github.com/mwanji/toml4j/issues/18

(a) Buggy and fixed code snippets in `emptyImplicitTable` method with commit message for the oracle bug-fixing commit

Title: Parsing exception messages contain trailing newlines

Utterance #1:
Some of the parsing exceptions thrown by toml4j contains trailing newlines. This is somewhat unusual, and causes empty lines in log files when the exception messages are logged...

Utterance #2:
The idea was to be able to display multiple error messages at once. However, processing stops as soon as an error is encountered, so that's not even possible. Removing the newlines shouldn't be a problem, then.

Solution Description: remove trailing newlines from toml4j log messages

(b) Bug-fixing patch from the toml4j project, with context from the corresponding bug report discussion.

Figure 8.1: Bug-fixing patch from the toml4j project, with context from the corresponding bug report discussion.

8.1 Motivation

Most existing approaches for automated bug fixing only consider the buggy code snippet when generating the fix. However, with such limited context, this is extremely challenging. For instance, in Figure 8.1a, generating the fixed code requires removing `.append("\n")`, but this is not obvious from inspecting the buggy code alone. To address this, Chakraborty and Ray (2021) proposed prompting de-

velopers for a natural language description of intent (e.g., "Removed trailing newlines...") that can guide a model in performing the task. As a proxy, in their study, they used the commit message corresponding to the oracle commit which fixed the bug.

By showing that natural language can aid bug-fixing, their study yields promising results. However, we raise two concerns with their approach. First, prompting developers for additional information can be burdensome for them, as it requires time and manual effort. Second, and more importantly, it is unrealistic to use the oracle commit message as a proxy. Since it is written *after* the bug is fixed to document the code changes (Tao et al., 2021), it does not accurately reflect information actually available when the task needs to be performed.

In reality, there are more appropriate sources of natural language to guide fixing bugs, which are *naturally* occurring and available *before* the task is to be performed. Namely, many bugs are first reported through issue tracking systems (e.g., GitHub Issues), where developers engage in a *discussion* to collectively understand the problem, investigate the cause, and formulate a solution (before they are fixed) (Chapter 7).

Content in these discussions are often relevant to generating the fix. For example, in Figure 8.1b, the title suggests that the bug pertains to "trailing newlines" and the last utterance of the discussion recommends "removing the newlines." Additionally, using our technique (Chapter 7) that summarizes content relevant towards implementing the solution in a bug report discussion, we can also automatically obtain a natural language description of the solution ("remove trailing newlines...").

Note that these sequences provide insight on the intent of the fix, much like the oracle commit message, without requiring any additional input or any context beyond what is naturally available. In this work, we use bug report discussions to facilitate automated bug fixing.

8.2 Deriving Context from Bug Report Discussions

We devise various strategies for heuristically and algorithmically deriving context from bug report discussions.

8.2.1 *Heuristically* Deriving Context

We consider using the *whole discussion*, including the title and all utterances (occurring before the bug-fixing code changes are implemented). However, these discussions can be extremely long (Table 8.1), making them difficult for neural models to reason about and also extending beyond the input length capacities of many models (e.g., 1,024 tokens) (Ahmad et al., 2021) in some cases. For this reason, we look at more concise elements within the discussion which might convey its meaningful aspects. First, we consider the *title*, as it is a brief summary of the bug (Chen et al., 2020). Next, we consider the *last utterance* before the bug-fixing commit, since it captures the most recent information and also roughly corresponds to the point at which a developer acquired enough context about the fix to implement it (Section 7.2.1).

8.2.2 *Algorithmically* Deriving Context

To guide developers in absorbing information relevant towards implementing the solution for a given bug report, in Chapter 7, we proposed generating a brief natural language description of the solution by synthesizing relevant content from within the whole bug report discussion. We finetuned PLBART on a filtered dataset, to build PLBART (F), the best-performing model for generating *solution descriptions* (Table 7.4).

While these solution descriptions are intended to guide humans in manually fixing bugs, we evaluate whether they can also guide models in automatically performing the task. Furthermore, since the title corresponding to the bug report discussion and the solution description summarize different aspects of the discussion, we investigate the benefits of combining the two (solution description + title).

Next, the segments (title or individual utterances) from the discussion that contribute the most towards generating a natural language description of the solution are likely to also be useful towards implementing that solution (i.e., generating the fix). To approximate the most relevant discussion segments, we use attention. Namely, we examine the last layer of PLBART (F)'s decoder to determine the most highly attended input token at each decoding step and the segment (title or individual utterance) to which it belongs. From this, we obtain the *attended segments*.

8.3 Data

Chakraborty and Ray (2021) relied on the commonly used bug-fixing patches

(*BFP*) datasets (Tufano et al., 2019). This entails BFP_{small}, with examples extracted from Java methods spanning fewer than 50 tokens, and BFP_{medium}, with examples extracted from methods spanning 50-100 tokens. In this work, we also focus on these datasets, particularly the preprocessed versions released by Chakraborty and Ray (2021). However, since they do not include the associated bug report discussions, we enrich examples with this information.

8.3.1 Mining Bug Report Discussions

We mine issue reports from GitHub Issues, for the 58,597 projects that encompass examples in the *BFP* datasets. We obtain 1,878,096 issue reports, 365,005 of which are linked to commits made between March 2011 and October 2017 (time frame used for mining the *BFP* datasets). By matching these commits to the *bug-fixing* commits from which the *BFP* examples were drawn, we identify the examples that correspond to *bug* reports. We map 3,028 (of the 58,287) examples in BFP_{small} and 3,333 (of the 65,404) examples in BFP_{medium} to bug report discussions, forming the *discussion-augmented bug-fixing patches* (*Disc-BFP*) datasets: $Disc\text{-}BFP_{small}$ and $Disc\text{-}BFP_{medium}$.

Note that *Disc-BFP* is comparatively smaller than *BFP*. While constructing *BFP*, Tufano et al. (2019) did not consider any mining criteria related to bug reports, so it is not surprising that many of their examples do not have bug report discussions. Bugs can be identified through various development activities like code review, testing, and bug reporting. In this work, we focus on the last scenario, for which bug report discussions would naturally be available.

	***Disc-BFP*$_{small}$**	***Disc-BFP*$_{med}$**
#Ex	3,028	3,333
#Discussions/Ex	1.3	1.3
#Utterance/Discussion	2.8	2.9
#Attn Segments/Ex	1.0	1.0
Buggy	22.1	42.4
Fixed	19.3	40.8
Method	32.2	74.2
Oracle Msg	19.7	19.6
Title	7.9	8.1
Utterance	127.6	136.4
Last Utterance	114.0	109.3
Soln Desc	8.5	8.5

Table 8.1: ***Disc-BFP* dataset statistics**. We report averages across all data splits. Average token lengths (split by punctuation and spacing) are presented in the second block. Note that we consider only utterances occurring before the bug-fixing commit.

Disc-BFP$_{small}$ consists of 2,445 training, 290 validation, and 293 test examples. *Disc-BFP*$_{medium}$ consists 2,660 training, 341 validation, and 332 test examples. In doing this, we maintain the original data splits (e.g., *Disc-BFP*$_{small}$'s training set is strictly a subset of *BFP*$_{small}$'s training set).

8.3.2 Data Processing

A bug report discussion is organized as a timeline (Section 7.2.1), and we consider only content that precedes the bug-fixing commit on the timeline, corresponding to the naturally-available context. Since a commit can be linked to multiple issue reports, some examples have multiple bug report discussions. In these cases, we order them so that discussions with the most recent activity appear first and are less likely to get truncated due to input length constraints (as explained in the next paragraph). When leveraging individual discussion components (e.g., title, generated solution description), we derive them from each discussion separately

and concatenate them (separated with <s>).

Though PLBART is capable of handling up to 1,024 tokens as input, Chakraborty and Ray (2021) limit to 512 tokens. However, since the sequences we consider can be particularly long after the SentencePiece tokenization (Kudo and Richardson, 2018) employed by PLBART, we choose to utilize the full capacity during finetuning. Note that the input is truncated by removing from the end if it exceeds the limit.

Before using PLBART (F) to obtain solution descriptions, we re-train the model after removing 7 examples in the original training set (Table 7.1) that have bug reports overlapping with the *Disc-BFP* test sets. We run inference on all partitions of the *Disc-BFP* datasets. For this, we first preprocess the bug report discussions following the original procedure (Section 7.2.3). Note that this preprocessing of bug report discussions is done only for generating solution descriptions. For our main models (Section 8.4), we rely solely on SentencePiece tokenization, closely following Chakraborty and Ray (2021). Additionally, bug report discussions often include source code, either in-lined with natural language or as longer code blocks, which are often delimited with markdown tags. In Section 7.2.3, we had retained in-lined code but removed longer marked blocks of code. While these longer code blocks may not be as relevant to generating natural language descriptions, we believe they could be useful in gathering insight for generating the fixed code. Therefore, we do not remove them from bug report discussions, even when generating solution descriptions. We provide dataset statistics in Table 8.1.

8.4 Models

Chakraborty and Ray (2021) achieved state-of-the-art performance on the *BFP* datasets by finetuning PLBART on large amounts of source code from GitHub and technical text from StackOverflow. Similarly, we consider finetuning PLBART to generate the fixed code given varying input context representations.

Since Chakraborty and Ray (2021) finetuned using significantly more data (i.e., *BFP* training sets), we initialize models using their checkpoints, corresponding to two different input context representations. The first one corresponds to concatenating the buggy code snippet and the full method context (`emptyImplicitTable` in Figure 8.1a): *buggy* <s> *method*. This helps contextualize the buggy code snippet and was shown to improve performance.[1] Since this representation entails only source code, we also consider the checkpoint for the representation that includes natural language: *buggy* <s> *method* <s> *oracle commit message*.[2]

8.4.1 Our Models

After initializing, we further finetune on the *Disc-BFP*$_{small}$ and *Disc-BFP*$_{medium}$ training sets (separately). All input context representations used for this are formed by concatenating *buggy* <s> *method* <s> with the various natural language sequences tied to bug report discussions outlined in Section 2.4. Sequences entailing multiple elements (e.g., utterances in the whole discussion, titles from multiple bug

[1] Note that the method also contains the buggy code snippet. Though repetitive, this outperformed a unified format.

[2] This input is used only for the initial finetuning and not used in the later finetuning stage or evaluation of our models.

report discussions) are separated with <s>.

8.4.2 Baselines

We consider models which use only *buggy* <s> *method* (without natural language). We also consider models that use the oracle commit message rather than context from bug report discussions: *buggy* <s> *method* <s> *oracle commit message*. We finetune baselines on the *Disc-BFP* training sets, using a context window of 1,024 tokens.

8.5 Results

Following Chakraborty and Ray (2021), we compute how often (%) the generated output *exactly matches* the target fixed code snippet. We perform statistical significance testing with bootstrap tests (Berg-Kirkpatrick et al., 2012), using 10,000 samples (with sample size 5,000) and $p < 0.05$.

We present results in Table 8.2. Reaffirming previous findings (Chakraborty and Ray, 2021) on the benefits of using natural language for fixing bugs, we find that leveraging context from bug report discussions yields from 1.8-5.4% improvement over baselines which do not include natural language context. Importantly, using bug report discussions leads to 1.5-3.0% improvements over baselines that use the oracle commit message (during finetuning and test). This suggests that context derived from bug report discussions, encompassing diverse types of information, can offer richer context than oracle commit messages for fixing bugs. This is especially promising since these discussions are often readily available in a real world setting.

Init	FT/Test Ctxt	*Disc-BFP$_{small}$*	*Disc-BFP$_{med}$*
Without NL (*BFP*)	Without NL*	33.8	27.1§¶
	Oracle Msg†	33.4	27.4§¶
	Whole Disc	33.1	27.1§¶
	Title	35.5*†	25.9
	Last Utterance	35.2*†	**28.9***†§¶
	Soln Desc	33.8	27.4§¶
	Soln Desc + Title	35.5*†	25.6
	Attended Seg	36.2*†	28.0*§¶
With NL (*BFP*)	Without NL§	35.5*†	25.3
	Oracle Msg¶	36.2*†	25.9
	Whole Disc	34.1	25.6
	Title	35.2*†	25.3
	Last Utterance	36.2*†	25.6
	Soln Desc	33.4	26.5§
	Soln Desc + Title	**39.2***†§¶	26.2§
	Attended Seg	36.9*†§	24.1

Table 8.2: **Results on the *Disc-BFP* test sets.** Models are initialized from one of the two checkpoints originally finetuned on the full *BFP* training sets, *without* and *with* NL. We then finetune on the *Disc-BFP* training sets with various input context representations and evaluate on the *Disc-BFP* test sets using the same representations. We indicate representations that statistically significantly outperform baselines with symbols.

Overall, the scores and magnitude of improvement tend to be lower for *Disc-BFP$_{medium}$*. This is likely due to the challenges of generating longer sequences (Varis and Bojar, 2021) and the stringent evaluation metric requiring exact match with the reference. The best performance on the *Disc-BFP$_{small}$* test set comes from using solution description + title. For *Disc-BFP$_{medium}$*, it is with the last utterance. Since both of these are derived from the whole discussion, one may expect using the whole discussion to yield similar or even improved performance; however, this is not the case.

Including the whole discussion substantially increases the input length, which models like PLBART cannot easily handle. This can be partially attributed to the

```
- public void assertEquals (java.lang.Object actual, java.lang.Object expected)
+ public void assertEquals (java.lang.Object expected, java.lang.Object actual)
{
   if ((expected == null) && (actual == null))
       return;
   if ((expected != null) && (expected.equals(actual)))
       return;
   fail(format(expected, actual));
}
```

Oracle Commit Message: Fixes issue #4.

Title: assertEquals parameters order

Utterance #1:
Maybe this is not an issue but a desired behaviour, however, it seems to me that the order of the parameters in the assertEquals method is wrong: public void assertEquals(Object actual, Object expected) Being a long time user of JUnit, I expected the "actual" parameter to be in the second position instead of the first one.

Utterance #2:
The ordering was based on TestNG, which is what I typically use for unit testing, but since xUnit is more common I don't mind reversing the order.

Solution Description: reversing the order of the assert equals parameters

Figure 8.2: Examples from the *Disc-BFP*$_{medium}$ test set, with the corresponding bug report discussion (`https://github.com/jhalterman/concurrentunit/issues/4`) and generated solution description.

practical challenge of fitting the entire sequence in the model's limited context window, with 12.8-15.8% training examples getting truncated. However, the bigger challenge is drawing meaning from such large amounts of text. We demonstrate the benefits of using more concise sequences, through various natural language elements that are likely to capture critical aspects of the whole discussion.

8.6 Examples

For the *Disc-BFP*$_{small}$ test example in Figure 8.1, the two models which leverage only *buggy* <s> *method* during finetuning and test (Without NL in Table 8.2) do not generate the correct output. Note that neither of these models have access to any natural language context. Two other models (which do use natural language) also fail to generate the correct output, corresponding to the whole discussion and solution description representations (initialized using the "Without NL"

Init	Finetune Context	*Disc-BFP$_{small}$*	*Disc-BFP$_{med}$*
Without NL (*BFP*)	Whole Discussion	36.9	29.8
	Title	40.3	27.4
	Last Utterance	36.9	**32.2**
	Attended Segments	37.2	31.3
With NL (*BFP*)	Whole Discussion	39.6	29.2
	Title	38.2	26.8
	Last Utterance	39.2	29.5
	Attended Segments	**42.3**	27.1

Table 8.3: Evaluating exact match (%) if the best performing segment (title or any individual utterance) from the whole discussion is used at test time (assuming that it's known).

checkpoint). In all four of these error cases, the model simply copies the buggy code snippet. However, the other 12 models generate the correct output for this particular example.

Some examples are difficult for models, even with natural language context. We provide one such example from the *Disc-BFP$_{medium}$* test set in Figure 8.2. The fix requires reversing the order of the method parameters, which is actually evident from the bug report discussion, as well as the generated solution description. However, performing this reversal involves more complex reasoning, and so the majority of models are unable to generate the correct output for this example. Nonetheless, the model which leverages the last utterance (initialized using the "Without NL" checkpoint) does manage to generate the correct output.

8.7 Analysis: Identifying Useful Discussion Segments

We acquire context from bug report discussions in various ways, either *heuristically* (whole discussion, title, last utterance) or *algorithmically* (attended segments when generating solution descriptions). (Note that we do not include solution de-

scriptions in these groups since they do not actually appear within the bug report discussions.) As we saw in Table 8.2, using the whole discussion may not be beneficial, since models struggle to reason about large amounts of text. We show that we are able to achieve improved performance by selecting more concise segments from within this discussion (e.g., title, last utterance, attended segments) that are likely to be relevant to fixing the bug.

However, we may not always being selecting the most useful segment(s) yielding the best performance. The most useful segments may vary by example, and there could also be other utterances (beyond the title, last utterance, and attended utterances) that have relevant information.

Therefore, we also estimate the performance of an "oracle" upper-bound that employs the most useful segment as the natural language context. For this, we consider models finetuned with the various segments from the discussion, including models finetuned on the whole discussion. We run inference with these models, using *buggy* <s> *method* <s> *segment*, for all segments, including the title and each utterance in the discussion. So, if there are N segments derived from the discussion (title and $N-1$ utterances before the bug-fixing commit), we obtain N candidates for the fixed code.

For a given example, we compute *best exact match*, or how often *at least one* of these candidates matches the reference. We present results in Table 8.3. We observe a 3.1–3.3% gap, relative to the highest scores in Table 8.2, suggesting that there is useful context in these discussions that is not being exploited. We leave it to future work to learn models for extracting the most useful segments from bug

report discussions for fixing bugs.

8.8 Summary

In this work, we investigated the utility of natural language for automated bug fixing. Unlike prior work, which leveraged an unrealistic source of natural language for this purpose, through oracle commit messages, we considered a naturally occurring source that is often available: bug report discussions. We explored various strategies for deriving natural language context from these discussions, using our newly compiled discussion-augmented, bug-fixing patches datasets. We showed that when these discussions are available, they offer useful context for bug fixing, even leading to improved performance over using oracle commit messages.

8.9 Additional Details

Our models are based on the architecture of PLBART, which itself follows from the BART-base model (Lewis et al., 2020). The encoder and decoder each have 6 layers, with hidden dimension 768 and 12 heads. There are approximately 140M parameters. We use the same hyperparameters as Chakraborty and Ray (2021). The batch size is 4, with gradient accumulation over every 4 batches. Early stopping is employed, with a patience of 5 epochs, based on validation performance. All models are trained for a single run. At test time, beam search is used, with a beam size of 5.

In Table 8.2, we present results from initializing model parameters from two

Init	Context	Inference Only		Finetuned	
		Disc-BFP$_{small}$	***Disc-BFP$_{med}$***	***Disc-BFP$_{small}$***	***Disc-BFP$_{med}$***
PLBART	Without NL	-	-	22.2	14.8
	Oracle Msg	-	-	28.0	16.6
	Whole Disc	-	-	25.3	16.0
	Title	-	-	27.3	1.5
	Last Utterance	-	-	23.2	19.0
	Soln Desc	-	-	20.5	17.2
	Soln Desc + Title	-	-	24.2	16.3
	Attended Seg	-	-	18.1	1.8
Without NL (*BFP*)	Without NL	30.7	25.3	33.8	27.1
	Oracle Msg	30.7	25.0	33.4	27.4
	Whole Disc	21.5	19.0	33.1	27.1
	Title	31.4	25.9	35.5	25.9
	Last Utterance	29.0	23.2	35.2	**28.9**
	Soln Desc	31.7	25.9	33.8	27.4
	Soln Desc + Title	29.7	25.0	35.5	25.6
	Attended Seg	23.5	20.8	36.2	28.0
With NL (*BFP*)	Without NL	31.1	22.3	35.5	25.3
	Oracle Msg	31.1	24.4	36.2	25.9
	Whole Disc	20.5	16.9	34.1	25.6
	Title	28.7	22.3	35.2	25.3
	Last Utterance	25.3	22.3	36.2	25.6
	Soln Desc	29.4	24.1	33.4	26.5
	Soln Desc + Title	28.3	22.6	**39.2**	26.2
	Attended Seg	23.5	19.9	36.9	24.1

Table 8.4: We measure the effect of finetuning on the *Disc-BFP* training sets by comparing to a setting in which the Chakraborty and Ray (2021) checkpoints are used directly for inference (without any finetuning). We also measure the effect of initializing with checkpoints that have already been finetuned on task-specific data by comparing to models directly initialized from PLBART and then finetuned on the *Disc-BFP* training sets.

of the checkpoints released by Chakraborty and Ray (2021). One corresponds to finetuning PLBART *without NL* using task-specific data from the larger *BFP* training sets. The other one corresponds to finetuning PLBART *with NL* (from oracle commit messages), also using task-specific data from the *BFP* training sets. Since these checkpoints have already been finetuned on bug-fixing data, it is reasonable to run inference on them directly without further finetuning on the *Disc-BFP* training sets. We show these results in Table 8.4. We find the overall performances to be lower, especially when testing with input context representations that were not seen during Chakraborty and Ray (2021)'s finetuning (e.g., whole discussion).

We also tried initializing model parameters directly from PLBART and finetuning on the *Disc-BFP* training sets. Table 8.4 shows that this works poorly, likely because the *Disc-BFP* training sets are smaller than the *BFP* training sets, with which the Chakraborty and Ray (2021) checkpoints were finetuned. Therefore, to reap the benefits of finetuning on more data, we believe it is best to first finetune on larger bug-fixing datasets (for which bug report discussions do not need to be available). Following that, another stage of finetuning should be done using the smaller training set that includes context from bug report discussions.

Chapter 9

Future Work

In this chapter, we outline future directions for using natural language to *support* and *drive* software evolution (Sections 9.1-9.2). We also discuss our ideas for using natural language to learn improved source code representations for broader applications in software engineering (Section 9.3). Finally, we describe the steps required in applying the research presented in this thesis to real-world software development (Section 9.4).

9.1 Unifying Related Tasks Occurring Upon Code Changes

In this thesis, we frame supporting software evolution in terms of assisting developers with upholding software quality when they make code changes. For this, we focused on detecting and updating comments based on code changes. However, these are just two of the many development tasks that are performed following code changes. For example, developers must write commit messages to document the code changes in a given commit, write release notes to document changes in a set of commits, and update the test suite based on code changes to verify the modified functionality.

There have been efforts to automate these tasks (Loyola et al., 2017; Moreno et al., 2014; Mirzaaghaei, 2011). Note that these tasks are closely related, and the general idea is the same: *Given code changes, generate X.* However, researchers

tend to develop models that are very task-specific and evaluate models on those tasks only. This makes it difficult to truly appreciate a newly proposed model and also makes it harder to study new problems, since existing models could have constraints that are not applicable to different problems.

Recently, Chen et al. (2021b) proposed a unified framework that is suitable for addressing multiple tasks, such as code summarization, bug classification, and bug repair. Inspired by this, we propose applying a similar strategy with the various tasks that occur following code changes. For this, we could also consider a multi-task learning technique, in which tasks can complement one another during training. Furthermore, we could evaluate whether large pretrained autoregressive models, like Codex (Chen et al., 2021a) and PaLM (Chowdhery et al., 2022), are able to reason about code changes in such a way that they more or less offer a "unified framework" for addressing these tasks through prompt engineering and few shot learning.

9.2 Interactively Participating in Code Review Discussions

We frame the goal of driving code evolution in terms of expediting critical code changes. In this thesis, we focused on using bug report discussions to generate solution descriptions and suggested bug-fixing code snippets to guide developers in quickly resolving bugs. However, there is a second step in which the code changes must be *reviewed* by other developers to ensure that they efficiently implement the correct functionality and adhere to established style guidelines (Brown and Parnin, 2020; Li et al., 2017).

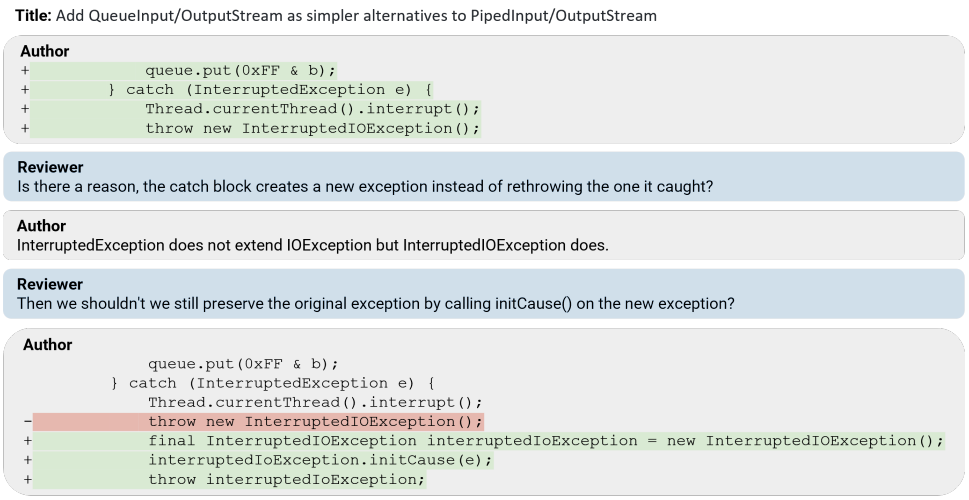


Figure 9.1: Code review discussion from the Apache Commons IO project: `https://github.com/apache/commons-io/pull/171`.

As illustrated in Figure 9.1, like bug report discussions, code review discussions entail interactive dialogue between developers. Reviewers post comments about specific parts of the code changes to point out problems they see and describe additional code changes for addressing these problems. The author of the code changes responds by posting comments or by 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).

Code review is a very time-consuming process that requires significant manual effort (Hellendoorn et al., 2021; Jiang et al., 2021a; Wessel et al., 2020). Due to developers' tight project schedules, code review and the release of important changes can get delayed (Yu et al., 2015; Maddila et al., 2020).

There have been efforts to build tools for streamlining reviewing through auxiliary tasks like automatically recommending relevant reviewers (Yu et al., 2014), review prioritization (van der Veen et al., 2015). There is also work that aims to more directly assist developers with code review by identifying parts of a given set of code changes that likely need to be reviewed, generating review comments, and refining code based on a particular review comment (Hellendoorn et al., 2021; Tufano et al., 2021; Li et al., 2022). However, all of these ignore the interactive nature of code review.

As future work, we propose building an agent that can participate in code review discussions and essentially collaborate with human developers for more efficient code review. The agent can assume two different roles. First, it can interactively provide review comments to guide developers in making code changes. Second, it can even participate in the discussion by generating suggested code changes based on reviewers, which may help the author more quickly address them.

We believe such an agent can be useful in settings beyond code review, such as an educational programming environment (Li and Boyer, 2016b), in which it can take on the role of a tutor by providing natural language feedback to students on their code and suggested code changes to improve code quality.

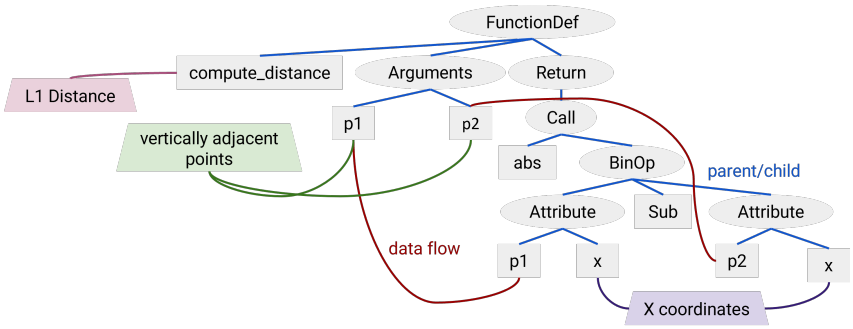
9.3 Enhancing Code Representations with Natural Language

As discussed in Section 2.5, there are various ways for representing source code for software-related tasks, such as with token sequences, abstract syntax trees (ASTs), and control and data flow graphs. However, the accompanying natural lan-

```
"""Computes the distance between two
vertically adjacent points as the L1
distance between their X coordinates."""

def compute_distance(p1, p2):
    return abs(p1.x - p2.x)
```

(a) Method and Comment



(b) Enhanced Code Representation

Figure 9.2: Illustration of an NL-enhanced code representation. In 9.2a, we show a method and its accompanying comment, with annotated spans that can be aligned to nodes (and edges) in the graph presented in 9.2b. The gray nodes in the graph correspond to AST nodes, with parent/child edges shown in blue. Red edges correspond to data flow edges.

guage elements (e.g., natural language comments), are often ignored when building these representations. For example, considering only the code in the method shown in Figure 9.2a, we can build a graph representation (Figure 9.2b) using AST nodes (gray nodes), AST edges (blue edges), and data flow edges (red edges). Using the AST nodes/edges, we see that $p1$ and $p2$ are both arguments of the method since they are children of the *Arguments* node. From the data flow edges, we also see that they both appear beneath the *Binary Operation* node. However, we do not obtain any obtain any additional information about these two entities.

However, from the comment, we know that the method assumes that $p1$ and $p2$ are *vertically adjacent points*, so this is another relation we can incorporate into

this representation. Similarly, the specific type of distance that is being computed is *L1 distance* based on the comment, and the x attributes of $p1$ and $p2$ can be further described as X coordinates from the comment. Therefore, by considering the natural language comment, we can enhance this representation with more fine-grained relation and type information which may be useful to a model.

We believe developing such NL-enhanced code representations is an interesting future direction, with potential implications for a diverse set of applications. Building these representations requires first extracting bimodal relationships between natural language and source code entities. While we explored this in Chapter 3, we focused on learning a binary relationship, on whether a natural language entity is associated with a particular entity in code, rather than on *how* it is associated. We leave it to future work to extract more fine-grained relations.

9.4 Applying Research to Real-World Software Development

In this thesis, we evaluated our models through automated metrics and simulated user studies. While these results are promising and suggest that our models can be useful towards streamlining software development activities, we have not yet been able to measure their true utility in a real-world environments. For example, we envision building an Integrated Development Environment (IDE) plugin that can detect and update comments as developers make code changes in real-time.[1] Additionally, we envision building custom GitHub tools in order to integrate our models for generating solution descriptions and suggested bug-fixes into the issue

[1]We could also evaluate how developers perceive the utility of this tool, compared to a tool which operates in the opposite direction (e.g., given comment changes, generate code changes).

tracking workflow. Such tools will allow us to conduct a more accurate evaluation on how useful our models are in helping real developers in accomplishing tasks. Evaluating in such a setting also gives us an opportunity to improve our models by understanding the types of errors they are prone to making and also by collecting qualitative feedback from developers.

Building, deploying, and evaluating such tools require many steps. For instance, we will need to collaborate with Human-Computer Interaction (HCI) experts to determine exactly how our model predictions should be presented to developers, in such a way that they are most easily accessible without being distracting. Next, we must design a mechanism for quality control that ensures that we do not display low-quality or factually incorrect model predictions. To be able to quickly display model predictions to users, we must be able to achieve minimal model latency. Additionally, to address a diverse set of user needs, we need to scale up to more programming languages (beyond Java) and more types of data (e.g., class-level comments, in-line comments). Finally, we must determine the most appropriate metrics for evaluating performance, such as how often a generated suggestion is accepted or the average difference in time needed to perform a given task with and without our model predictions. Moreover, in addition to evaluating whether these tools expedite certain development activities, we could evaluate whether they improve the overall quality of software artifacts (e.g., comments), by measuring trends in the number of stars and forks.

Chapter 10

Conclusions

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 design tasks, models, and datasets to support and drive code evolution through natural language.

First, we studied natural language comments, and we developed a rich set of features for explicitly associating comments (Chapter 3). which we found to be useful for our later tasks. Next, 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 (Chapter 4). Additionally, 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 (Chapter 5). 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 (Chapter 6).

Furthermore, 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. We designed techniques for using these discussions to expedite bug resolution. First, to enable developers to more easily locate and comprehend information relevant towards implementing the bug-fixing code changes, we proposed an automated system which generates a concise natural language description of the solution as soon as the necessary context becomes available in an ongoing bug report discussion (Chapter 7). Using supervision derived from commits and pull requests, we benchmarked approaches for generating informative solution descriptions. We also conducted a study on integrating such a generation model into a real-time setting by pipelining or jointly training 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. Next, to provide developers with suggested bug-fixing code changes, we improved automated bug-fixing models by incorporating natural language context from generated solution descriptions as well as other elements from within bug report discussions (Chapter 8).

In Chapter 9, we outlined future directions for using natural language to support and drive code evolution. This includes developing a unified framework for addressing the many tasks that occur following code changes and building an interactive agent for automated code review. Additionally, we discussed a more

general future direction for enhancing the code representations used for different software-related tasks with natural language, to capture broader context and more fine-grained relation and type information. We also described a way forward in applying the research presented in this thesis to build developer tools for real-time settings.

In conclusion, this thesis makes important contributions to the growing field of research focused on designing machine learning and NLP techniques to facilitate software development activities. We believe our research has laid the groundwork for progress in this field which can lead to more efficient and effective software development, ultimately leading to more reliable software in all facets of life, in years to come.

References

Ibrahim Abdelaziz, Julian Dolby, James P McCusker, and Kavitha Srinivas. Graph4Code: A machine interpretable knowledge graph for code. *arXiv preprint arXiv:2002.09440*, 2020.

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

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

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

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

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

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

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

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

Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. Self-supervised bug detection and repair. *NeurIPS*, 34, 2021.

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

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

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

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

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

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

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

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

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

Muhammad Zubair Baloch, Shahid Hussain, Humaira Afzal, Muhammad Rafiq Mufti, and Bashir Ahmad. Software developer recommendation in terms of reducing bug tossing length. In Guojun Wang, Bing Chen, Wei Li, Roberto Di Pietro, Xuefeng Yan, and Hao Han, editors, *SpaCCS*, pages 396–407, 2021.

Satanjeev Banerjee and Alon Lavie. 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, 2005.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. SequenceR: Sequence-to-sequence learning for end-to-end program repair. *TSE*, 47(9):1943–1959, 2019.

Songqiang Chen, Xiaoyuan Xie, Bangguo Yin, Yuanxiang Ji, Lin Chen, and Baowen Xu. Stay professional and efficient: Automatically generate titles for your bug reports. In *ASE*, pages 385–397, 2020.

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. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Zimin Chen, Vincent J Hellendoorn, Pascal Lamblin, Petros Maniatis, Pierre-Antoine Manzagol, Daniel Tarlow, and Subhodeep Moitra. PLUR: A unifying, graph-based view of program learning, understanding, and repair. *NeurIPS*, 34:23089–23101, 2021.

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

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. PaLM: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

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

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

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

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

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

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

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

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

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

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

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

Günes Erkan and Dragomir R. Radev. LexRank: Graph-based lexical centrality as salience in text summarization. *Journal of Artificial Intelligence Research*, 22(1):457–479, 2004.

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

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

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

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

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

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

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

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

R. Stuart Geiger, Kevin Yu, Yanlai Yang, Mindy Dai, Jie Qiu, Rebekah Tang, and Jenny Huang. 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, 2020.

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

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

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

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

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

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

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

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

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

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

Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. CC2Vec: Distributed representations of code changes. In *ICSE*, pages 518–529, 2020.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. iFixR: Bug report driven program repair. In *ESEC/FSE*, pages 314–325, 2019.

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

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

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

Taku Kudo and John Richardson. SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *EMNLP: System Demonstrations*, pages 66–71, 2018.

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

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

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

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

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

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

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

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

Katherine Lee, Daphne Ippolito, Andrew Nystrom, Chiyuan Zhang, Douglas Eck, Chris Callison-Burch, and Nicholas Carlini. Deduplicating training data makes language models better. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8424–8445, 2022.

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

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

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

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

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

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

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

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

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

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

Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, et al. CodeReviewer: Pre-training for automating code review activities. *arXiv preprint arXiv:2203.09095*, 2022.

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

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

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

Yang Liu and Mirella Lapata. Text summarization with pretrained encoders. In *EMNLP-IJCNLP*, pages 3730–3740, 2019.

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

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

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

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

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

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

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

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

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

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

Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. CoCoNut: combining context-aware neural translation models using ensemble for program repair. In *ISSTA*, pages 101–114, 2020.

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

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

Ehsan Mashhadi and Hadi Hemmati. Applying CodeBERT for automated program repair of java simple bugs. In *MSR*, pages 505–509, 2021.

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

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

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

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

Mehdi Mirzaaghaei. Automatic test suite evolution. In *ESEC/FSE*, pages 396–399, 2011.

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

Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrian Marcus, and Gerardo Canfora. Automatic generation of release notes. In *FSE*, pages 484–495, 2014.

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

Dana Movshovitz-Attias and William W Cohen. Grounded discovery of coordinate term relationships between software entities. *arXiv preprint arXiv:1505.00277*, 2015.

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

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

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

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

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

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

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

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

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

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

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

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. A conversational paradigm for program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

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

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

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, et al. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114*, 2021.

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

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

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

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

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

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

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

Sheena Panthaplackel, Milos Gligoric, Junyi Jessy Li, and Raymond J. Mooney. Learning to describe solutions for bug reports based on developer discussions. *Under Submission*, 2022.

Sheena Panthaplackel, Junyi Jessy Li, Milos Gligoric, and Ray Mooney. Learning to describe solutions for bug reports based on developer discussions. In *Findings of ACL*, pages 2935–2952, 2022.

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

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

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

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

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

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Wei Tao, Yanlin Wang, Ensheng Shi, Lun Du, Shi Han, Hongyu Zhang, Dongmei Zhang, and Wenqiang Zhang. On the evaluation of commit message generation models: An experimental study. In *ICSME*, pages 126–136, 2021.

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

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

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

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

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

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

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

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

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

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

Dusan Varis and Ondřej Bojar. Sequence length is a domain: Length-based overfitting in transformer models. In *EMNLP*, pages 8246–8257, 2021.

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

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

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

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

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

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *EMNLP*, pages 8696–8708, 2021.

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

Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. How long will it take to fix this bug? In *MSR*, pages 1–1, 2007.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Vita

Sheena Panthaplackel was brought up in a suburb of Chicago, Illinois, and she graduated from Downers Grove South High School in 2013. For her undergraduate studies, she attended the University of Illinois at Urbana-Champaign, where she obtained a B.S. degree in Computer Science in 2016. Following that, she worked as a software developer at a trading firm in Chicago. In August 2017, she moved to Austin to begin the PhD program in the Department of Computer Science at the University of Texas at Austin.

She was a recipient of the Bloomberg Data Science Fellowship in 2020. Her research has been published as conference papers at AAAI 2020, ACL 2020, AAAI 2021, and ACL 2022. She has also been a reviewer at AAAI, EMNLP, ARR, and workshops at ICSE, AACL, and NAACL. During her PhD, she also did summer internships at Microsoft Research Cambridge and Bloomberg AI, where she worked on research that have also been published as conference papers in AAAI 2021 and ACL 2022.

Permanent Address: spantha@utexas.edu

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, Ayman El-Khashab and Dan Garette. The author used a template released by Dan Garette with small modifications to account for changes in formatting requirements.