# Learning to Update Natural Language Comments Based on Code Changes

**Sheena Panthaplackel**[1], **Pengyu Nie**[2], **Milos Gligoric**[2], **Junyi Jessy Li**[3], **Raymond J. Mooney**[1]
[1]Department of Computer Science
[2]Department of Electrical and Computer Engineering
[3]Department of Linguistics
The University of Texas at Austin
spantha@cs.utexas.edu, pynie@utexas.edu, gligoric@utexas.edu,
jessy@austin.utexas.edu, mooney@cs.utexas.edu



## Abstract

We formulate the novel task of automatically updating an existing natural language comment based on changes in the body of code it accompanies. We propose an approach that 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. We train and evaluate our model using a dataset that we collected from commit histories of open-source software projects, with each example consisting of a concurrent update to a method and its corresponding comment. We compare our approach against multiple baselines using both automatic metrics and human evaluation. Results reflect the challenge of this task and that our model outperforms baselines with respect to making edits.

## 1 Introduction

Software developers include natural language comments alongside source code as a way to document various aspects of the code such as functionality, use cases, pre-conditions, and post-conditions. With the growing popularity of open-source software that is widely used and jointly developed, the need for efficient communication among developers about code details has increased. Consequently, comments have assumed a vital role in the development cycle. With developers regularly refactoring and iteratively incorporating new functionality, source code is constantly evolving; however, the accompanying comments are not always updated to reflect the code changes (Tan et al., 2007; Ratol and Robillard, 2017). Inconsistency between code and comments can not only lead time-wasting confusion in tight project schedules (Hu et al., 2018) but can also result in bugs (Tan et al., 2007). To address this problem, we propose an approach that can automatically suggest comment updates when the associated methods are changed.

**Previous Version**

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

**Updated Version**

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

Figure 1: Changes in the `getRotX` method and its corresponding `@return` comment between two subsequent commits of the rajawali-rajawali project, available on GitHub.

Prior work explored rule-based approaches for detecting inconsistencies for a limited set of cases; however, they do not present ways to automatically fix these inconsistencies (Tan et al., 2007; Ratol and Robillard, 2017). Recent work in automatic comment generation aims to generate a comment given a code representation (Liang and Zhu, 2018; Hu et al., 2018; Fernandes et al., 2019); although these techniques could be used to produce a completely new comment that corresponds to the most recent version of the code, this could potentially discard salient content from the existing comment that should be retained. To the best of our knowledge, we are the first to formulate the task of *automatically updating an existing comment when the corresponding body of code is modified.*

This task is intended to align with how developers edit a comment when they introduce changes in the corresponding method. Rather than deleting it and starting from scratch, they would likely only modify the specific parts relevant to the code updates. For example, Figure 1 shows the `getRotX` method being modified to have the return value parsed into degrees. Within the same commit, the corresponding comment is revised to indicate this, without imposing changes on parts of the comment that pertain to other aspects of the return value. 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. Namely, our model

is trained to generate a sequence of *edit actions*, which are to be applied to the existing comment, by conditioning on learned representations of the code edits and existing comment. We additionally incorporate linguistic and lexical features to guide the model in determining where edits should be made in the existing comment. Furthermore, we develop an output reranking scheme that aims to produce edited comments that are fluent, preserve content that should not be changed, and maintain stylistic properties of the existing comment.

We train and evaluate our system on a corpus constructed from open-source Java projects on GitHub, by mining their commit histories and extracting examples from consecutive commits in which there was a change to both the code within a method as well as the corresponding Javadoc comment, specifically, the `@return` Javadoc tag. These comments, which have been previously studied for learning associations between comment and code entities (Panthaplackel et al., 2020), follow a well-defined structure and describe characteristics of the output of a method. For this reason, as an initial step, we focus on `@return` comments in this work. Our evaluation consists of several automatic metrics that are used to evaluate language generation tasks as well as tasks that relate to editing natural language text. We also conduct human evaluation, and assess whether human judgments correlate with the automatic metrics.

The main contributions of this work include (1) the task of automatically updating an existing comment based on source code changes and (2) a novel approach for learning to relate edits between source code and natural language that outperforms multiple baselines on several automatic metrics and human evaluation. Our implementation and data are publicly available.[1]

## 2 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 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.

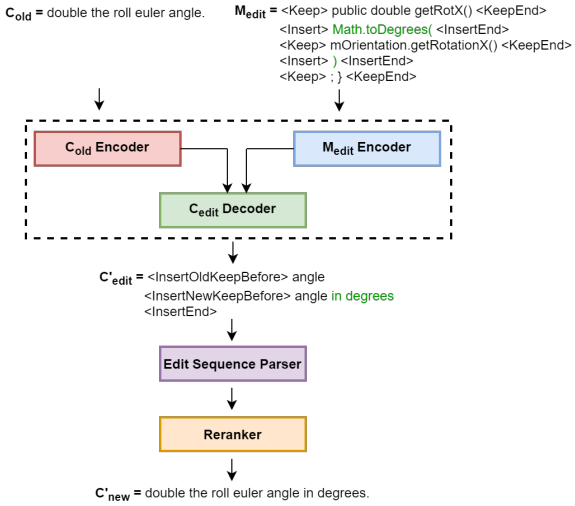


Figure 2: High-level overview of our system.

## 3 Edit Model Overview

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. Since $C_{old}$ and $C_{new}$ are closely related, training a model to directly generate $C_{new}$ risks having it learn to just copy $C_{old}$. To explicitly inform the model of edits, we define the target output as a *sequence of edit actions*, $C_{edit}$, to indicate how the existing comment should be revised (e.g., for $C_{old}$=`ABC`, $C_{edit}$=`<Delete>A<DeleteEnd>` implies that `A` should be deleted to produce $C_{new}$=`BC`). Furthermore, in order to better correlate these edits with changes in the code, we unify $M_{old}$ and $M_{new}$ into a single *diff* sequence that explicitly identifies code edits, $M_{edit}$. We discuss in more detail how $M_{edit}$ and the training $C_{edit}$ are constructed in §4.

Figure 2 shows a high-level overview of our system. We design an encoder-decoder architecture consisting of three components: a two-layer, bidirectional GRU (Cho et al., 2014) that encodes the code changes ($M_{edit}$), another two-layer, bidirectional GRU that encodes the existing comment ($C_{old}$), and a GRU that is trained to decode a sequence of edit actions ($C_{edit}$).[2] We concatenate the

[1] https://github.com/panthap2/LearningToUpdateNLComments

[2] We refrain from using the self-attention model (Vaswani et al., 2017) because prior work (Fernandes et al., 2019) suggests that it yields lower performance for comment generation.

final states of the two encoders to form a vector that summarizes the content in $M_{edit}$ and $C_{old}$, and use this vector as the initial state of the decoder. 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 (§4.4).

# 4 Representing Edits

Here we define the edit lexicon that is used to construct the input code edit sequence, $M_{edit}$, and the target comment edit sequence, $C_{edit}$.

## 4.1 Edit Lexicon

We use difflib[3] to extract code edits and target comment edits. Both the input code edit sequence and the target comment edit sequence consist of a series of edit actions; each edit action is structured as `<Action> [span of tokens] <ActionEnd>`.[4]

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>
```

## 4.2 Code Edits

We extract the edits between $M_{old}$ and $M_{new}$ using the edit lexicon to construct $M_{edit}$, the code edit sequence used as input in one of the encoders. Figure 2 (top right) shows the $M_{edit}$ corresponding to code changes in Figure 1.

In contrast to line-level code *diffs* that are commonly used for commit message generation (Loyola et al., 2017; Jiang et al., 2017; Xu et al., 2019), this representation allows us to explicitly capture more fine-grained edits. While we could exploit the abstract syntax tree (AST) structure of source code and represent the changes between the ASTs corresponding to the two versions of code, prior work suggests that such techniques do not always lead to improved performance (Yin et al., 2019). We leave it to future work to investigate how the AST structure can be leveraged for this task.

## 4.3 Comment Edits

We identify the changes between $C_{old}$ and $C_{new}$ to construct $C_{edit}$, the target comment edit sequence. During inference, the output comment is produced by parsing the predicted edit sequence (§4.4). We introduce a slightly modified set of specifications that disregards the `Keep` type when constructing the sequence of edit actions, referred to as the *condensed edit sequence*.

The intuition for disregarding `Keep` and the span of tokens to which it applies is that we can simply copy the content that is retained between $C_{old}$ and $C_{new}$, instead of generating it anew. By doing post-hoc copying, we simplify learning for the model since it has to only learn *what to change* rather than also having to learn *what to keep*.

We design a method to deterministically place edits in their correct positions in the absence of

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

[4] 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., 2018; Dong et al., 2019; Awasthi et al., 2019).

`Keep` spans. For the example in Figure 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 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. See Appendix A for details on this procedure.

### 4.4 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. 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 Features

We extract linguistic and lexical features for tokens in $M_{edit}$ and $C_{edit}$, many of which were shown to improve learning associations between `@return` comment and source code entities in our prior work (Panthaplackel et al., 2020). We incorporate these features into the network as one-hot vectors that are concatenated to $M_{edit}$ and $C_{edit}$ embeddings and then passed through a linear layer. These vectors are provided as inputs to the two encoders. All sequences are subtokenized, e.g., `camelCase` → `camel`, `case`.

**Features specific to $M_{edit}$:** We aim to take advantage of common patterns among different types of code tokens 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. Finally, we include a feature to indicate whether the token matches a token in $C_{old}$. This is intended to help the model identify locations in $M_{edit}$ that may be relevant to editing $C_{old}$.

**Features specific to $C_{old}$:** We include whether a token matches a code token that is inserted, deleted, or replaced in $M_{edit}$. These help align parts of $C_{old}$ with code edits, assisting the model in determining where edits should be made. In order to exploit common patterns for different types of tokens, we incorporate features that identify whether the token appears more than once in $C_{old}$ or is a stop word, and its part-of-speech.

**Shared features:** We include whether the token is a subtoken that was originally part of a larger token and its index if so (e.g., split from `camelCase`, `camel` and `case` are subtokens with indices 0 and 1 respectively). These features aim to encode important relationships between adjacent tokens that are lost once the body of code and comment are transformed into a single, subtokenized sequences. Additionally, because we focus on `@return` comments, we introduce features intended to guide the model in identifying relevant tokens in $M_{edit}$ and $C_{old}$. Namely, we include whether a given token matches a token in a `return` statement that is unique to $M_{old}$, unique to $M_{new}$, or present in both. Similarly, we indicate whether the token matches a token in the subtokenized `return` type that is unique to $M_{old}$, unique to $M_{new}$, or present in both.

## 6 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 rescore the candidates: 1) generation likelihood and 2) similarity to $C_{old}$. These heuristics are computed after parsing the candidate edit sequences (§4.4).

**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 pretrained comment generation model (§8.2) that is

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

Table 1: Number of examples, projects, and edit actions; average similarity between $M_{old}$ and $M_{new}$ as the ratio of overlap; average similarity between $C_{old}$ and $C_{new}$ as the ratio of overlap; 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.

trained on a substantial amount of data for generating $C_{new}$ given only $M_{new}$. We compute the length-normalized probability of this model generating the parsed candidate comment, C'$_{new}$, (i.e., $P(C'_{new}|M_{new})^{1/N}$ 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.[5]

**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).

## 7 Data

We extracted *examples* from popular, open-source Java projects using GitHub's commit history. We extract pairs of the form (method, comment) for the same method across two consecutive commits where there is a simultaneous change to both the code and comment. This creates somewhat noisy data for the task of comment update; Appendix B describes filtering techniques to reduce this noise.

[5]We attempted to integrate this model into the training procedure of the edit model through joint training; however, this deteriorated performance.

We first tokenize $M_{old}$ and $M_{new}$ using the javalang[6] library. We subtokenize based on camelCase and snake_case, as in previous work (Allamanis et al., 2016; Alon et al., 2019; Fernandes et al., 2019). We then form $M_{edit}$ from the subtokenized forms of $M_{old}$ and $M_{new}$. We tokenize $C_{old}$ and $C_{new}$ by splitting by space and punctuation. We remove HTML tags and the "@return" that precedes all comments, and also subtokenize tokens since code tokens may appear in comments as well. The gold edit action sequence, $C_{edit}$, is computed from these processed forms of $C_{old}$ and $C_{new}$.

To avoid having examples that closely resemble one another in training and test, the projects in the training, test, and validation sets are disjoint, similar to Movshovitz-Attias and Cohen (2013). Table 1 gives dataset statistics. Of the 7,239 examples in our final dataset, 833 of them were extracted from the diffs used in Panthaplackel et al. (2020). Including code and comment tokens that appear at least twice in the training data as well as the predefined edit keywords, the code and comment vocabulary sizes are 5,945 and 3,642 respectively.

## 8 Experimental Method

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

### 8.1 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.

[6]https://pypi.org/project/javalang/

### 8.2 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, bi-directional GRU 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. We expect the incorporation of more complicated architectures, e.g., tree-based (Alon et al., 2019) and graph-based (Fernandes et al., 2019) encoders which exploit AST structure, can be applied to both an edit model and a generation model, which we leave for future work.

Evaluation is based on the 736 ($M_{new}$, $C_{new}$) pairs in the test set described in §7. We ensure that the projects from which training examples are extracted are disjoint from those in the test set.

### 8.3 Reranked Generation Model

In order to allow the generation model to exploit the old comment, this system uses similarity to $C_{old}$ (cf. §6) 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).

### 8.4 Model Training

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 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.

## 9 Evaluation

### 9.1 Automatic Evaluation

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

Previous work suggests that BLEU-4 fails to accurately capture performance for tasks related to edits, such as text simplification (Xu et al., 2016), grammatical error correction (Napoles et al., 2015), and style transfer (Sudhakar et al., 2019), since a system that merely copies the input text often achieves a high score. Therefore, we also include two text-editing metrics to measure how well our system learns to *edit*: SARI (Xu et al., 2016), originally proposed to evaluate text simplification, is essentially the average of N-gram F1 scores corresponding to add, delete, and keep edit operations;[7] GLEU (Napoles et al., 2015), used in grammatical error correction 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.

**Results:** We report automatic metrics averaged across three random initializations for all learned models, and use bootstrap tests (Berg-Kirkpatrick et al., 2012) for statistical significance. Table 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

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

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

Table 2: Exact match, METEOR, BLEU-4, SARI, and GLEU scores. Scores for which the difference in performance is *not* statistically significant ($p < 0.05$) are indicated with matching symbols.

performance with respect to SARI; however, it provides statistically significant improvements on most other metrics.

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

### 9.2 Human Evaluation

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

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

| Baseline | Generation | Edit | None |
|---|---|---|---|
| 18.4% | 12.4% | 30.2% | 55.0% |

Table 3: Percentage of annotations for which users selected comment suggestions produced by each model. All differences are statistically significant ($p < 0.05$).

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

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

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

**Results:** Table 3 presents the percentage of annotations (out of 500) for which users selected

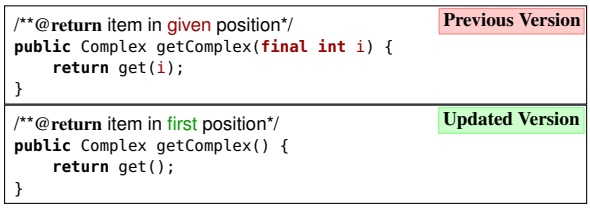


Figure 3: Changes in the `getComplex` method and its corresponding `@return` comment between two subsequent commits of the eclipse-january project, available on GitHub.

comment suggestions that were produced by each model. Using Krippendorff's $\alpha$ (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 these, we observe that in a large portion of these cases, the comment did not warrant an update. This is consistent with prior work in sentence simplification which shows that, very often, there are sentences that do not need to be simplified (Li and Nenkova, 2015). Despite our efforts to minimize such cases in our dataset through rule-based filtering techniques, we found that many remain. This suggests that it would be beneficial to train a classifier that first determines whether a comment needs to be updated before proposing a revision. Furthermore, the cases for which the existing comment does need to be updated but none of the models produce reasonable predictions illustrate the scope for improvement for our proposed task.

## 10 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 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 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. Sample outputs are given in Appendix C.

## 11 Ablations

We empirically study the effect of training the network to encode explicit code edits and decode explicit comment edits. As discussed in Section 3, 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 4.

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 features introduced in §5. As shown in Table 5, these features improve performance by wide margins, across all metrics.

| Inputs | Output | xM (%) | METEOR | BLEU-4 | SARI | GLEU |
|---|---|---|---|---|---|---|
| $C_{old}$, $M_{new}$ | $C_{new}$ | 5.707$^{\ddagger\P}$ | 29.259$^{\dagger}$ | 33.534$^{\S}$ | 28.024 | 30.000$^{*}$ |
| | $C_{edit}$ | 4.755$^{\ddagger *}$ | 33.796 | 43.315 | 35.516 | 37.970$^{\|}$ |
| $C_{old}$, $M_{old}$, $M_{new}$ | $C_{new}$ | 3.714$^{*}$ | 18.729 | 20.060 | 23.914 | 21.956 |
| | $C_{edit}$ | 5.163$^{\ddagger\P}$ | 34.895 | 44.006$^{*}$ | 33.479 | 37.618$^{\|}$ |
| $C_{old}$, $M_{edit}$ | $C_{new}$ | 6.114$^{\P}$ | 29.968$^{\dagger}$ | 34.164$^{\S}$ | 28.980 | 30.491$^{*}$ |
| | $C_{edit}$ | **8.922** | **36.229** | **44.283**$^{*}$ | **40.538** | **39.879** |

Table 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.

| | Model | xM (%) | METEOR | BLEU-4 | SARI | GLEU |
|---|---|---|---|---|---|---|
| Models | Edit | 17.663 | 42.222 | 48.217 | **46.376** | 45.060 |
| | - feats. | 8.922$^{\dagger}$ | 36.229 | 44.283 | 40.538 | 39.879$^{*}$ |
| Reranked models | Edit | **18.433** | **44.698** | **50.717** | 45.486 | **46.118** |
| | - feats. | 8.877$^{\dagger}$ | 38.446 | 46.665 | 36.924 | 40.317$^{*}$ |

Table 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.

## 12 Related Work

**Learning from source code changes:** Lee et al. (2019) use rule-based techniques to automatically detect and revise outdated API names in code documentation; however, their approach cannot be extended to full natural language comments that are the focus of this work. Zhai et al. (2020) propose a technique for updating incomplete and buggy comments by propagating comments from different code elements (e.g., variables, methods, classes) based on program analysis and several heuristics. Rather than simply copying a related comment, we aim to revise an outdated comment by reasoning about code changes. Yin et al. (2019) present an approach for learning structural and semantic properties of source code edits so that they can be generalized to new code inputs. Similar to their work, we learn vector representations from source code changes; however, unlike their setting, we apply these representations to natural language. Prior work in automatic commit message generation aims to learn from code changes in order to generate a natural language summary of these changes (Loyola et al., 2017; Jiang et al., 2017; Xu et al., 2019). Instead of generating natural language content from scratch as done in their work, we focus on applying edits to existing natural language text. We also show that generating a comment from scratch does not perform as well as our proposed edit model for the comment update setting.

**Editing natural language text:** Approaches for editing natural language text have been studied extensively through tasks such as sentence simplification (Dong et al., 2019), style transfer (Li et al., 2018), grammatical error correction (Awasthi et al., 2019), and language modeling (Guu et al., 2018). The focus of this prior work is to revise sentences to conform to stylistic and grammatical conventions, and it does not generally consider broader contextual constraints. On the contrary, our goal is not to make cosmetic revisions to a given span of text, but rather amend its semantic meaning to be in sync with the content of a separate body of information on which it is dependent. More recently, Shah et al. (2020) proposed an approach for rewriting an outdated sentence based on a sentence stating a new factual claim, which is more closely aligned with our task. However, in our case, the separate body of information is not natural language and is generally much longer than a single sentence.

## 13 Conclusion

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.

## Acknowledgements

We thank reviewers for their feedback on this work and the participants of our user study for their time. This work was partially supported by a Google Faculty Research Award and the US National Science Foundation under Grant Nos. CCF-1652517 and IIS-1850153.

## References


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

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

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating sequences from structured representations of code. In *International Conference on Learning Representations*.

Fernando Alva-Manchego, Louis Martin, Carolina Scarton, and Lucia Specia. 2019. EASSE: Easier automatic sentence simplification evaluation. In *Conference on Empirical Methods in Natural Language Processing and International Joint Conference on Natural Language Processing: System Demonstrations*, pages 49–54.

Abhijeet Awasthi, Sunita Sarawagi, Rasna Goyal, Sabyasachi Ghosh, and Vihari Piratla. 2019. Parallel iterative edit models for local sequence transduction. In *Conference on Empirical Methods in Natural Language Processing and International Joint Conference on Natural Language Processing*, pages 4251–4261.

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

Taylor Berg-Kirkpatrick, David Burkett, and Dan Klein. 2012. An empirical investigation of statistical significance in NLP. In *Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 995–1005.

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

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

Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Structured neural summarization. In *International Conference on Learning Representations*.

Kelvin Guu, Tatsunori B Hashimoto, Yonatan Oren, and Percy Liang. 2018. Generating sentences by editing prototypes. *Transactions of the Association for Computational Linguistics*, 6:437–450.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *International Conference on Program Comprehension*, pages 200–210.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Annual Meeting of the Association for Computational Linguistics*, pages 2073–2083.

Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically generating commit messages from diffs using neural machine translation. In *International Conference on Automated Software Engineering*, pages 135–146.

Wei-Jen Ko, Greg Durrett, and Junyi Jessy Li. 2019. Linguistically-informed specificity and semantic plausibility for dialogue generation. In *North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 3456–3466.

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

Reno Kriz, João Sedoc, Marianna Apidianaki, Carolina Zheng, Gaurav Kumar, Eleni Miltsakaki, and Chris Callison-Burch. 2019. Complexity-weighted loss and diverse reranking for sentence simplification. In *North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 3137–3147.

Seonah Lee, Rongxin Wu, S.C. Cheung, and Sungwon Kang. 2019. Automatic detection and update suggestion for outdated API names in documentation. *Transactions on Software Engineering*.

Juncen Li, Robin Jia, He He, and Percy Liang. 2018. Delete, retrieve, generate: a simple approach to sentiment and style transfer. In *North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1865–1874.

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

Yuding Liang and Kenny Q. Zhu. 2018. Automatic generation of text descriptive comments for code blocks. In *AAAI Conference on Artificial Intelligence*, pages 5229–5236.

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

*Empirical Methods in Natural Language Processing*, pages 2122–2132.

Pablo Loyola, Edison Marrese-Taylor, and Yutaka Matsuo. 2017. A neural architecture for generating natural language descriptions from source code changes. In *Annual Meeting of the Association for Computational Linguistics*, pages 287–292.

Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective approaches to attention-based neural machine translation. In *Conference on Empirical Methods in Natural Language Processing*, pages 1412–1421.

Dana Movshovitz-Attias and William W. Cohen. 2013. Natural language models for predicting programming comments. In *Annual Meeting of the Association for Computational Linguistics*, pages 35–40.

Courtney Napoles, Keisuke Sakaguchi, Matt Post, and Joel Tetreault. 2015. Ground truth for grammatical error correction metrics. In *Annual Meeting of the Association for Computational Linguistics and the International Joint Conference on Natural Language Processing*, pages 588–593.

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

Sheena Panthaplackel, Milos Gligoric, Raymond J. Mooney, and Junyi Jessy Li. 2020. Associating natural language comment and source code entities. In *AAAI Conference on Artificial Intelligence*.

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

Rebecca Passonneau. 2006. Measuring agreement on set-valued items (MASI) for semantic and pragmatic annotation. In *International Conference on Language Resources and Evaluation*.

Inderjot Kaur Ratol and Martin P. Robillard. 2017. Detecting fragile comments. *International Conference on Automated Software Engineering*, pages 112–122.

Darsh J. Shah, Tal Schuster, and Regina Barzilay. 2020. Automatic fact-guided sentence modification. In *AAAI Conference on Artificial Intelligence*.

Richard Shin, Illia Polosukhin, and Dawn Song. 2018. Towards specification-directed program repair. In *International Conference on Learning Representations Workshop*.

Akhilesh Sudhakar, Bhargav Upadhyay, and Arjun Maheswaran. 2019. "Transforming" delete, retrieve, generate approach for controlled text style transfer. In *Conference on Empirical Methods in Natural Language Processing and the International Joint Conference on Natural Language Processing*, pages 3267–3277.

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

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer networks. In *Advances in Neural Information Processing Systems*, pages 2692–2700.

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

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

Ziyu Yao, Daniel S. Weld, Wei-Peng Chen, and Huan Sun. 2018. StaQC: A systematically mined question-code dataset from Stack Overflow. In *International Conference on World Wide Web*, pages 1693–1703.

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

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

Juan Zhai, Xiangzhe Xu, Yu Shi, Guanhong Tao, Minxue Pan, Shiqing Ma, Lei Xu, Weifeng Zhang, Lin Tan, and Xiangyu Zhang. 2020. CPC: Automatically classifying and propagating natural language comments via program analysis. In *International Conference on Software Engineering*.

| | 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 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.

# A Modified Comment Edit Lexicon

We first transform insertions and ambiguous deletions into a structure that resembles Replace, characterized by InsertOld/InsertNew and DeleteOld/DeleteNew spans respectively. 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 6 shows statistics on how often each of these edit actions are used. We present more details about the actions in the sections that follow.

## A.1 Replacements

**Replace** This action is defined as shown below:

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

It 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 corresponding $C_{edit}$ 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** This action is defined as shown below:

```
<ReplaceOldKeepBefore>[span of old tokens]
<ReplaceNewKeepBefore>[span of new tokens]
<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. Therefore, to represent editing $C_{old}$=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, which appears before the edit location.

**ReplaceKeepAfter** This action is defined as shown below:

```
<ReplaceOldKeepAfter>[span of old tokens]
<ReplaceNewKeepAfter>[span of new tokens]
<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.

### A.2 Insertions

We disregard basic `Insert` actions since it is always ambiguous where an insertion should occur without an anchor point. Following what is done for ambiguous `Replace` actions, we introduce `InsertKeepBefore` and `InsertKeepAfter`.

**`InsertKeepBefore`** This action is defined as shown below:

```
<InsertOldKeepBefore>[span of old tokens]
<InsertNewKeepBefore>[span of new tokens]
<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`** This action is defined as shown below:

```
<InsertOldKeepAfter>[span of old tokens]
<InsertNewKeepAfter>[span of new tokens]
<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.

### A.3 Deletions

**`Delete`** This action is defined as shown below:

```
<Delete>[span of old tokens]<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`** This action is defined as shown below:

```
<DeleteOldKeepBefore>[span of old tokens]
<DeleteNewKeepBefore>[span of new tokens]
<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`** This action is defined as shown below:

```
<DeleteOldKeepAfter>[span of old tokens]
<DeleteNewKeepAfter>[span of new tokens]
<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 given below:

```
<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.

## B Data Filtering

As done in Panthaplackel et al. (2020), we apply heuristics to reduce the number of cases in which the code and comment changes are unrelated. First, because we focus on `@return` comments that pertain to the return values of a given method, we discard any example in which the code change does not entail either a change to the return type or at least one return statement. Then, to identify the correct mapping of two versions of a method among other changes in a commit, we focus on the code changes that preserve the method names. It may happen sometimes that developers change the method name as well as code and comment in one commit, but we leave it as future work to improve this filtering heuristic. Next, we attempt to remove examples in which the comment change appears to be purely stylistic (e.g. spelling corrections, reformatting, and rephrasing). Furthermore, prior work (Allamanis, 2019) has shown that duplication can adversely affect evaluation of machine learning models for code and language tasks. For this reason, we remove duplicates from our corpus.

Despite having mined commit histories for thousands of projects, upon filtering, we are left with a total of 7,239 examples belonging to 1,081 different projects. This demonstrates the challenge of collecting large datasets with relatively low levels of noise in this domain. Although online code resources like GitHub and StackOverflow host large quantities of data that can be exploited for transduction tasks between source code and natural language, prior work has shown that much of this data is unusable without cleaning (Yin et al., 2018).

Some have used rule-based techniques to do data cleaning (Allamanis et al., 2016; Hu et al., 2018; Fernandes et al., 2019), and others train classifiers on hand-labeled examples that can be applied to a much larger pool of examples in order to differentiate between clean and noisy examples (Iyer et al., 2016; Yao et al., 2018; Yin et al., 2018). Most of these approaches focus on code summarization or comment generation which only require single code-NL pairs for training and evaluation as the task entails generating a natural language summary of a given code snippet. On the contrary, our proposed task requires two code-NL pairs that are assumed to hold specific parallel relationships with one another. Namely, the relationship between $C_{new}$ and $M_{new}$ is expected to be similar to that of $C_{old}$ and $M_{old}$. The relationship between $C_{new}$ and

$C_{old}$ is expected to correlate with the relationship between $M_{new}$ and $M_{old}$. Not only does having four moving parts in one example magnify noise, but the need to hold these relationships makes data cleaning particularly difficult. We leave building classifiers for aiding this process as future work.

## C Sample Output

In Table 7, we show predictions for various examples in the test set.

## 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 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).