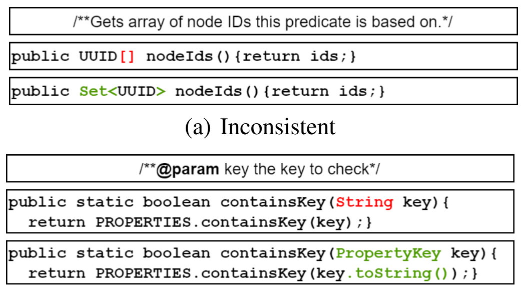
Deep Just-In-Time Inconsistency Detection Between Comments and Source Code

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

Abstract

Natural language comments convey key aspects of source code such as implementation, usage, and pre- and post-conditions. Failure to update comments accordingly when the corresponding code is modified introduces inconsistencies, which is known to lead to confusion and software bugs. In this paper, we aim to detect whether a comment becomes inconsistent as a result of changes to the corresponding body of code, in order to catch potential inconsistencies *just-in-time*, i.e., before they are committed to a code base. To achieve this, we develop a deep-learning approach that learns to correlate a comment with code changes. By evaluating on a large corpus of comment/code pairs spanning various comment types, we show that our model outperforms multiple baselines by significant margins. For extrinsic evaluation, we show the usefulness of our approach by combining it with a comment update model to build a more comprehensive automatic comment maintenance system which can both detect and resolve inconsistent comments based on code changes.



```
/**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 1: In the example from the Apache Ignite project shown in Figure 1(a), the existing comment becomes inconsistent upon changes to the corresponding method, and in the example from the Alluxio project shown in Figure 1(b), the existing comment remains consistent after code changes.

1 Introduction

Comments serve as a critical communication medium for developers, facilitating program comprehension and code maintenance tasks (**??**). Code is highly-dynamic in nature, with developers constantly making changes to address bugs and feature requests. 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 (**???**). Therefore, it is desirable to have systems that can automatically detect such inconsistencies and alert developers.

Previous work has explored heuristic-based approaches for automatically detecting specific types of inconsistencies (e.g., identifier naming (**?**), parameter constraints (**?**), `null` values and exceptions (**?**), locking (**?**), interrupts (**?**)). Some have also addressed the notion of coherence between comments and code as a text similarity problem with traditional machine learning models that leverage bag-of-words techniques (**??**). In contrast, we design an approach that generalizes across types of inconsistencies and captures deeper comment/code relationships. Furthermore, prior research has predominantly focused on detecting inconsistencies that already reside in a software project, within the code repository. 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 (**?**), 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 base. We refer to this as *just-in-time inconsistency detection*, as it allows catching potential inconsistencies right before they can materialize.

Detecting inconsistencies immediately following code changes allows us to utilize information from the version of the code before the changes, for which the comment is consistent. 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 1(a), the comment describes the return type of `nodeIds()` as an 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.

Moreover, due to challenges in crafting data extraction rules (**??**) and annotating substantial amounts of data (**?**), prior post hoc work relies on a limited set of examples and projects. In contrast, we build a large corpus for just-in-time inconsistency detection by mining commit histories of software projects for code changes with and without corresponding comment updates.

Few approaches exploit code changes for inconsistency detection and these rely on task-specific rules (**?**), hand-engineered surface features (**??**), and bag-of-words techniques (**?**). Instead, we *learn* salient characteristics of these inputs through a deep-learning framework that encodes their syntactic structures. Namely, we use recurrent neural networks (RNNs) and gated graph neural networks (GGNNs) (**?**) to learn contextualized representations of the comment and code changes and multi-head attention (**?**) to relate these representations in order to discern how the code changes affect the comment. We also study how manual features can complement our neural approach.

Furthermore, on its own, an inconsistency detection system can only flag comments that developers failed to update. Actually amending them to reflect code changes requires significant developer effort. Approaches for automatically updating comments based on code changes have been recently proposed (**??**). However, they do not handle cases in which an update is not needed, such as in Figure 1(b). While the type of the `key` argument is modified, its purpose is unchanged (i.e., it still represents the key to be checked in `PROPERTIES`). Based on our user study (**?**), such cases deteriorated the overall quality of the system. As a form of extrinsic evaluation, we evaluate the utility of our approach by integrating it with this comment update model, to build a more comprehensive automatic comment maintenance system that detects and resolves inconsistencies.

To summarize, our main contributions are as follows: (1) We develop a deep learning approach for just-in-time inconsistency detection that correlates a comment with changes in the corresponding body of code and which outperforms the post hoc setting as well as several baselines. (2) For training and evaluation, we construct a large corpus of comments paired with code changes in the corresponding methods, encompassing multiple types of method comments and consisting of 40,688 examples that are extracted from 1,518 open-source Java projects.[1] (3) We demonstrate the value of inconsistency detection in a comprehensive automatic comment maintenance system, and we show how our approach can support such a system.

[1]Data and implementation are available at `https://github.com/panthap2/deep-jit-inconsistency-detection`.

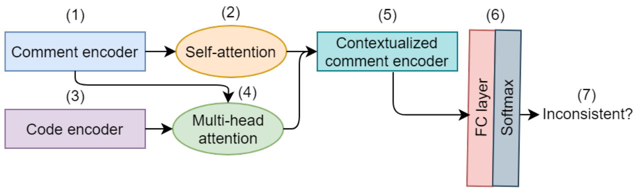


Figure 2: High-level architecture of our approach.

2 Task

Our task is to determine whether a comment is inconsistent, or semantically out of sync with the corresponding method. Most inconsistencies result from developers making code changes without properly updating the accompanying comments. Suppose M_{old} from the consistent comment/method pair (C, M_{old}) is modified to M. If C is not in sync with M and is not updated, it will become inconsistent once M is committed. We frame this problem in two distinct settings, with the task being constant across both: determine whether C is inconsistent with M.

- **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. Unlike the post hoc setting, M_{old} is available, allowing us to analyze the changes between M_{old} and M.

In line with most prior work in inconsistency detection (**????**), we focus on identifying inconsistencies in comments comprising API documentation for Java methods. API documentation consists of a main description and a set of tag comments (**?**). While some have considered treating the full documentation as a single comment (**?**), we choose to perform inconsistency detection at a more fine-grained level, analyzing individual comment types within this documentation. Furthermore, in contrast to previous studies tailored to a specific tag (**??**) or specific keywords and templates (**??**), 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, derived from the first sentence of the main description.

3 Architecture

We aim to determine whether C is inconsistent by understanding its semantics and how it relates to M (or changes between M_{old} and M). We show an overview of our approach in Figure 2. First, the comment encoder, a BiGRU (**?**), encodes the sequence of tokens in C (Figure 2 (1)). When learning a representation for a given token, the forward and backward BiGRU passes, in principle, provide context of other tokens in C. However, this information can get diluted, especially when there are long-range dependencies, and the relevant context can also vary across tokens. To

```
<Keep> public static boolean containsKey ( <KeepEnd>
<ReplaceOld> String <ReplaceNew> PropertyKey <ReplaceEnd>
<Keep> key ) { return PROPERTIES . containsKey ( key <KeepEnd>
<Insert> . toString ( ) <InsertEnd>
<Keep> ) ; } <KeepEnd>
```

Figure 3: Sequence-based code edit representation (M_{edit}) corresponding to Figure 1(b), with removed tokens in red and added tokens in green.

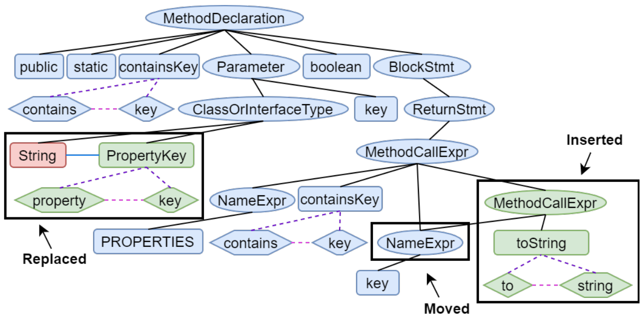


Figure 4: AST-based code edit representation (M_{edit}) corresponding to Figure 1(b), with removed nodes in red and added nodes in green.

address this, we update these representations from the comment encoder with more context about how they relate to the other tokens through multi-head self-attention (?) (Figure 2 (2)). Next, we learn code representations with a code encoder (Figure 2 (3)), which can be a sequence encoder (cf. §3.1) or an abstract syntax tree (AST) encoder (cf. §3.2).

Since the essence of the task comes down to whether C accurately reflects M, we must capture the relationship between C and M (or changes between M_{old} and M). Prior work does this by computing comment/code similarity through lexical overlap rules (**??**), 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 (**??**). 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. Therefore, their representations may not be very similar to the representation of the full method. In contrast, we learn the relationship between comments and code by computing multi-head attention between each hidden state of the comment encoder and the hidden states of the code encoder (Figure 2 (4)).

We combine the context vectors resulting from both attention modules to form enhanced representations of the tokens in C, which carry context from other parts of C as well as the code. These are then passed through another BiGRU encoder (Figure 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 2 (6)). This leads to the final prediction (Figure 2 (7)).

3.1 Sequence Code Encoder

In the just-in-time setting, we represent the changes between M_{old} and M with an edit action sequence, M_{edit}. We have previously shown that explicitly defining edits in such a way outperforms having the model implicitly learn them (**?**). Each action consists of an action type (`Insert`, `Delete`, `Keep`, `ReplaceOld`, `ReplaceNew`) that applies to a span of tokens, as shown in Figure 3. We encode M_{edit} with a BiGRU. Because M_{old} is unavailable in the post hoc setting, we cannot construct an edit action sequence. So, we encode the sequence of tokens in M.

3.2 AST Code Encoder

To better exploit the syntactic structure of code, we leverage its abstract syntax tree (AST). Following prior work in other tasks (**??**), we encode ASTs and AST edits using gated graph neural networks (GGNNs) (**?**). For the post hoc setting, we encode T, an AST-based representation corresponding to M. In the just-in-time setting, we instead encode T_{edit}, an AST-based edit representation. We use GumTree (**?**), to 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.

GGNN encoders for T and T_{edit} use *parent* (`public` $\rightarrow$ `MethodDeclaration`) and *child* (`MethodDeclaration` $\rightarrow$ `public`) edges. Like prior work (**?**), we add "subtoken nodes" for identifier leaf nodes to better handle previously unseen identifier names. To integrate these new nodes, we add *subnode* (`toString` $\rightarrow$ `to`), *supernode* (`to` $\rightarrow$ `toString`), *next subnode* (`to` $\rightarrow$ `string`), and *previous subnode* (`string` $\rightarrow$ `to`) edges. When encoding T_{edit}, we also include an *aligned* edge type between nodes in the two trees that correspond to an update (`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 Data

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). In prior work, we isolate comment updates made based on code changes through cases in which $C_1 \neq C_2$ (**?**). 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=C_1, M_{old}=M_1, and M=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

	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 1: Data partitions.

comment examples. We additionally incorporate 7,239 positive @`return` examples from our prior work (**?**) which studies @`return` comment updates.

While convenient for data collection, the assumptions we make do not always hold in practice. For instance, if C_1 is refactored without altering its meaning, we would assign a positive label because $C_1 \neq C_2$, despite it actually being consistent. Because such cases of *comment improvement* are not within the scope of our work, we adopt previously proposed heuristics (**?**) to reduce the number of instances in which the comment and code changes are unrelated. The negative label is also noisy since $C_1 = C_2$ when a developer fails to update comments in accordance with code changes, pointing to the problem we are addressing in this paper. We minimize such cases by limiting to popular, well-maintained projects (**?**). For more reliable evaluation, we curate a clean sample of 300 examples (corresponding to 101 projects) from the test set, consisting of 50 positive and 50 negative examples of each comment type.

In line with prior work (**??**), 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 (**?**). We downsample negative examples, for each partition and comment type, to construct a balanced dataset. Statistics of our final dataset are shown in Table 1.

Comments are tokenized based on space and punctuation. We parse methods into sequences using javalang (**?**). Comment and code sequences are subtokenized (e.g., camelCase $\rightarrow$ camel, case; snake_case $\rightarrow$ snake, case), as done in prior work (**??**), to capitalize on composability and better address the open vocabulary problem in learning from source code (**?**). Details on data statistics, filtering, and annotation procedures are given in Appendix **??**.

5 Models

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

5.1 Baselines

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

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

CodeBERT BOW: We develop a more sophisticated bag-of-words baseline that leverages CodeBERT (**?**) embeddings. These embeddings were pretrained on a large corpus of natural language/code pairs. In the post hoc setting, we consider CodeBERT BOW(C, M), which computes the average embedding vectors of C and M. 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, and we refer to this baseline as CodeBERT BOW(C, M_{edit}).

? (?): 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. Details about our re-implementation are given in Appendix **??**.

5.2 Our Models

Post hoc: We consider three models, with different ways of encoding the method. SEQ(C, M) encodes M with a GRU, GRAPH(C, T) encodes T with a GGNN, and HYBRID(C, M, T) uses both. Multi-head attention in HYBRID(C, M, T) is computed with the hidden states of the two encoders separately and then combined.

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.

Just-In-Time + features: Because injecting explicit knowledge can boost the performance of neural models (**??**), we investigate adding comment and code features to our approach. These are computed at the token/node-level and concatenated with embeddings before being passed to encoders. Features are derived from prior work on comments and code (**??**), including linguistic (e.g., POS tags) and lexical (e.g., comment/code overlap) features.

5.3 Model Training

Models are trained to minimize negative log likelihood. We use 2-layer BiGRU 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 (**?**). Edit embeddings are of dimension 8. Attention modules use 4 attention heads. We use a dropout

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

Table 2: Results for baselines, post hoc, and just-in-time models. Differences in F1 and Acc between just-in-time vs. baseline models, just-in-time vs. post hoc models, and just-in-time + features vs. just-in-time models are statistically significant.

	Model	Cleaned Test Sample				Full Test Set			
		P	R	F1	Acc	P	R	F1	Acc
`@return`	SEQ(C, M_{edit}) + features	88.5*	72.0*	**79.4***	**81.3***	**87.6***	73.3*	79.8*	**81.4***
	GRAPH(C, T_{edit}) + features	81.2	**77.3**	79.1*	79.7	82.2	**79.3**	**80.6**	80.9*
	HYBRID(C, M_{edit}, T_{edit}) + features	**88.7***	72.0*	**79.4***	**81.3***	87.3*	73.7*	79.8*	**81.4***
`@param`	SEQ(C, M_{edit}) + features	90.0	**95.3**	92.5	92.3†	92.2	88.3†	90.2	90.4
	GRAPH(C, T_{edit}) + features	**96.5**	92.0	**94.2**	**94.3**	**94.5**	**89.0**†	**91.7**	**91.9**
	HYBRID(C, M_{edit}, T_{edit}) + features	94.6	89.3	91.8	92.0†	93.3	85.9	89.4	89.9
Summary	SEQ(C, M_{edit}) + features	**96.0**	78.7	86.5§	87.7	84.7§	58.3	69.0	**73.9**§
	GRAPH(C, T_{edit}) + features	80.8	**92.0**	86.0§	85.0	76.0	**66.4**	**70.6**	72.5
	HYBRID(C, M_{edit}, T_{edit}) + features	93.7	86.0	**89.5**	**90.0**	**85.0**§	57.0	68.1	73.5§

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

rate of 0.6. Training ends if the validation F1 does not improve for 10 epochs.

6 Intrinsic Evaluation

We report common classification metrics: precision (P), recall (R), and F1 (w.r.t. the positive label) and accuracy (Acc), averaged across 3 random restarts. We also perform significance testing (**?**).

In Table 2, we report results for baselines, post hoc and just-in-time inconsistency detection models. In the post hoc setting, we find that our three models can achieve higher F1 scores than the bag-of-words approach proposed by **?** (**?**); however, they underperform the CodeBERT BOW(C, M) baseline and significantly underperform all just-in-time models, including the simple rule-based baseline. This demonstrates the benefit of performing inconsistency detection in the just-in-time setting, in which the code changes that trigger inconsistency are available. Additionally, by encoding the syntactic structures of the comment and code changes, our just-in-time models outperform this rule-based baseline as well as all other baselines and post hoc approaches. While the HYBRID(C, M_{edit}, T_{edit}) model achieves slightly higher scores (on the basis of F1 and accuracy) than SEQ(C, M_{edit}) and GRAPH(C, 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 (**????**) and feature sets (**?**) to build improved systems for specific domains.

Furthermore, in Table 3, we analyze the performance of the three just-in-time + features models with respect to individual comment types. While these models are trained on all comment types together without explicitly tailoring it in any way to handle them differently, they are still able to achieve reasonable performance across types. We provide further analysis of individual comment types and compare to comment-specific baselines in Appendix **??**.

7 Extrinsic Evaluation

We further evaluate how our approach could be used to build a comprehensive *just-in-time comment maintenance system* which first determines whether a comment, C, has become inconsistent upon code changes to the corresponding method ($M_{old} \rightarrow M$), and then automatically suggests an update if this is the case. To do this, we combine the inconsistency detection approach with our previously proposed comment update model (?) which updates comments based on code changes. For training and evaluating this combined system, we have two sets of comment/method pairs from consecutive commits for each example in our corpus. Recall from our data collection procedure that we extracted pairs of the form (C_1, M_1), (C_2, M_2), where C=C_1, M_{old}=M_1, and M=M_2. We now introduce C_{new}=C_2, the gold comment for M. If C is consistent with M, C=C_{new}.

7.1 Evaluation Method

The GRU-based Seq2Seq update model encodes C and a sequential representation of the code changes (M_{edit}). Using attention (?) and a pointer network (?) over learned representations of the inputs, a sequence of edit actions (C_{edit}) is generated, identifying how C should be edited to form the updated comment (C_{new}). This model also employs the same linguistic and lexical features as the just-in-time + features models. The model is trained on only cases in which C has to be updated and is not designed to ever copy the existing comment. We consider three different configurations for adding inconsistency detection in this model:

Update w/ implicit detection: We augment training of the update model with negative examples (i.e., C does not need to be updated). The model implicitly does inconsistency detection by learning to copy C for such cases. Inconsistency detection is evaluated based on whether it predicts C_{new}=C.

Pretrained update + detection: The update model is **?**, trained on only positive examples. At test time, if the detection model classifies C as inconsistent, we take the prediction of the update model. Otherwise, we copy C, making C_{new}=C. We consider three of the pretrained just-in-time detection models.

Jointly trained update + detection: We jointly train the inconsistency detection and update models on the full dataset (including positive and negative examples). We consider 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, we mask the loss of the update component since it does not have to learn to copy C. At test time, if the detection component predicts a negative label, we directly copy C and otherwise take the prediction of the update model.

7.2 Results

We report precision, recall, F1, and accuracy for detection. As we have done previously (?), we evaluate update through exact match (xMatch) as well as metrics used to evaluate text generation (BLEU-4 (?) and METEOR (?)) and text editing tasks (SARI (?) and GLEU (?)). In Table 4, 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 as C_{new}) and **?** (?), which is designed to always update (and only copy C if an invalid edit action sequence is generated). For completeness, we also provide results on the full dataset (which are analogous) in Appendix **??**.

Since our dataset is balanced, we can get 50% exact match by simply copying C (i.e., never updating). In fact, this can even beat **?** (?) on xMatch, METEOR, BLEU-4, SARI, and GLEU. This underlines the importance of first determining whether a comment needs to be updated, which can be addressed with our inconsistency detection approach. 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 **?** (?) is designed to *always* edit, it can perform well on this metric; however, the majority of the 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 as the prediction for C_{new} as a proxy for detecting inconsistency. It achieves high precision, but it frequently copies C 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 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. Overall, we find that our approach can be useful for building a real-time comment maintenance system. Since this is not the focus of our paper but rather merely a potential use case, we leave it to future work for developing more intricate joint systems.

8 Related Work

Code/Comment Inconsistencies: Prior work analyze how inconsistencies emerge (????) and the various types of inconsistencies (?); but, they do not propose techniques for addressing the problem.

Post Hoc Inconsistency Detection: Prior work propose rule-based approaches for detecting pre-existing inconsistencies in specific domains, including locks (?), interrupts (?), `null` exceptions for method parameters (??), and renamed identifiers (?). The comments they consider are consequently constrained to certain templates relevant to their respective domains. We instead develop a general-purpose, machine learning approach that is not catered towards any specific types of inconsistencies or comments. **?** (?) and **?** address a broader notion of coherence between comments and code through text-similarity tech-

	Update Metrics					Detection Metrics			
	xMatch	METEOR	BLEU-4	SARI	GLEU	P	R	F1	Acc
Never Update	50.0	67.4	72.1	24.9	68.2	0.0	0.0	0.0	50.0
? (**?**)	25.9	60.0	68.7	42.0*	67.4	54.0	**95.6**	69.0	57.1
Update w/ implicit detection	58.0	72.0	74.7	31.5	72.7	**100.0**	23.3	37.7	61.7
Pretrained update + detection									
SEQ(C, M_{edit}) + features	**62.3**†	75.6*	77.0*	42.0*	76.2	91.3*	82.0§	86.4*	87.1§¶
GRAPH(C, T_{edit}) + features	59.4	74.9§	76.6†	**42.5**$^{\|}$	75.8*†	85.8	87.1	86.4*	86.3†
HYBRID(C, M_{edit}, T_{edit}) + features	**62.3**†	75.8$^{\dagger\|}$	**77.2**	42.3†	**76.4**	92.3	82.4§	87.1†	87.8$^{*\|}$
Jointly trained update + detection									
SEQ(C, M_{edit}) + features	61.4*	**75.9**$^{\|}$	76.6†	42.4$^{\dagger\|}$	75.6†	88.3†	86.2	87.2†	87.3$^{\S\|}$
GRAPH(C, T_{edit}) + features	60.8	75.1§	76.6†	41.8*	75.8*	88.3†	84.7*	86.4*	86.7†¶
HYBRID(C, M_{edit}, T_{edit}) + features	61.6*	75.6*†	76.9*	42.3†	75.9*	90.9*	84.9*	**87.8**	**88.2***

Table 4: Results on joint inconsistency detection and update on the cleaned test sample. Scores for which the difference in performance is *not* statistically significant are shown with identical symbols.

niques, and **?** determine whether comments, specifically `@return` and `@param` comments, conform to particular format. We instead capture deeper code/comment relationships by learning their syntactic and semantic structures. **?** propose a siamese network for correlating comment/code representations. In contrast, we aim to correlate comments and code through an attention mechanism. **Just-In-Time Inconsistency Detection:** **?** (**?**) 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. **?** 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. **?** 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. **?** develops at approach which locates inconsistent identifiers upon code changes through lexical matching rules. While we find such a rule-based approach (represented by our OVERLAP(C, deleted) baseline) to be effective, a learned model performs significantly better. **?** 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. **?** 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.

9 Conclusion

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. We further conducted an extrinsic evaluation in which we demonstrated that our approach can be used to build a comprehensive comment maintenance system that can detect and update inconsistent comments.

Acknowledgments

This work was supported by the Bloomberg Data Science Fellowship and a Google Faculty Research Award.

Ethics Statement

Through this work, we aim to reduce time-consuming confusion and vulnerability to software bugs by keeping developers informed with up-to-date-documentation, in order to consequently help improve developers productivity and software quality. Buggy software and incorrect API usage can result in significant malfunctions in many everyday operations. Maintaining comment/code consistency can help prevent such negative-impact events. However, over-reliance on such a system could result in developers giving up identifying and resolving inconsistent comments themselves. By presuming that the system detects all inconsistencies and all of these are properly addressed, developers may also take the available comments for granted, without carefully analyzing their validity. Because the system may not catch all types of inconsistencies, this could potentially exacerbate rather than resolve the problem of inconsistent comments. Our system is not intended to serve as an infallible safety net for poor software engineering practices but rather as a tool that complements good ones, working alongside developers to help deliver reliable, well-documented software in a timely manner.

Possimus reiciendis illo ex quibusdam consequatur pariatur perspiciatis, nobis voluptatem impedit officia aspernatur, ipsam soluta pariatur voluptate inventore vitae ducimus.Fuga minus pariatur nulla asperiores maxime eos earum quos placeat eligendi, atque est iure facilis dolorem sunt consectetur earum?Et alias voluptatem placeat, adipisci nulla dignissimos inventore tenetur