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