Automating the Removal of Obsolete TODO Comments

Zhipeng Gao
Monash University
Australia
zhipeng.gao@monash.edu

Xin Xia*
Monash University
Australia
xin.xia@acm.org

David Lo
Singapore Management University
Singapore
davidlo@smu.edu.sg

John Grundy
Monash University
Australia
john.grundy@monash.edu

Thomas Zimmermann
Microsoft Research
United States
tzimmer@microsoft.com

ABSTRACT

TODO comments are very widely used by software developers to describe their pending tasks during software development. However, after performing the task developers sometimes neglect or simply forget to remove the TODO comment, resulting in obsolete TODO comments. These obsolete TODO comments can confuse development teams and may cause the introduction of bugs in the future, decreasing the software's quality and maintainability. Manually identifying obsolete TODO comments is time-consuming and expensive. It is thus necessary to detect obsolete TODO comments and remove them automatically before they cause any unwanted side effects. In this work, we propose a novel model, named TDCleaner (**T**O**D**O comment **Cleaner**), to identify obsolete TODO comments in software projects. TDCleaner can assist developers in just-in-time checking of TODO comments status and avoid leaving obsolete TODO comments. Our approach has two main stages: offline learning and online prediction. During offline learning, we first automatically establish $\langle code\_change, todo\_comment, commit\_msg \rangle$ training samples and leverage three neural encoders to capture the semantic features of TODO comment, code change and commit message respectively. TDCleaner then automatically learns the correlations and interactions between different encoders to estimate the final status of the TODO comment. For online prediction, we check a TODO comment's status by leveraging the offline trained model to judge the TODO comment's likelihood of being obsolete. We built our dataset by collecting TODO comments from the top-10,000 Python and Java Github repositories and evaluated TDCleaner on them. Extensive experimental results show the promising performance of our model over a set of benchmarks. We also performed an in-the-wild evaluation with real-world software projects, we reported 18 obsolete TODO comments identified by TDCleaner to Github developers and 9 of them have already been confirmed and removed by the developers, demonstrating the practical usage of our approach.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**.

KEYWORDS

TODO comment, Obsolete comment, Code-Comment Inconsistency, Code-comment co-evolution, BERT model

ACM Reference Format:
Zhipeng Gao, Xin Xia, David Lo, John Grundy, and Thomas Zimmermann. 2021. Automating the Removal of Obsolete TODO Comments. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3468264.3468553

1 INTRODUCTION

TODO comments in source code are extensively used by developers to denote their pending tasks. After completing (some of) the documented pending tasks, developers should update or remove their associated TODO comment(s). Such an example is shown in Ex.1 of Fig 1. Here a developer added a TODO comment (Line 87, *TODO: check rackspace file existence*) to notify themselves or others of the unfinished task. When the developer (or someone else in the team) updated the source code (highlighted in green colour) to perform this task, the accompanying TODO comment was also deleted (highlighted in red colour). However, due to time constraints or carelessness, developers may have completed (or partially completed) the task specified by the TODO comment but forget to remove it (or update it) [33, 39]. This results in TODO comments becoming obsolete and more and more irrelevant and unreliable when the software changes and evolves.

In this work, we define a TODO comment as an *obsolete TODO comment* if its corresponding task is accomplished but the TODO comment itself is not removed. These obsolete TODO comments lay down outdated tasks that need not, or should not, be followed any longer. A good TODO comment can help developers in understanding the designed tasks as well as the source code [2, 41]. On the contrary, an obsolete TODO comment can obscure the source code and affect code comprehension. Previous studies [14, 25, 33] have shown that developers lose confidence in the reliability of the system when they encountered outdated comments. Moreover, once obsolete TODO comments get separated from their code, they

*This is the corresponding author

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3468553

Ex.1 Commit Message: Check for file existence on Rackspace

```
87  -       # TODO: Check rackspace file existence
85  +       if isinstance(uploader, rackspace.RackspaceUploader):
86  +           return uploader.file_exists(filename, self._container(app))
87  +       return True
```

Ex.2 Commit Message: convert lambda to class to reduce coupling

```
148 -                   final DecoratingMember<T> mCopy = m;
149 -                   // TODO convert lambda to class to decouple from `this`
150 -                   scheduler.scheduleDirect(() -> mCopy.release());
148 +                   scheduler.scheduleDirect(new Releaser<T>(m));
```

Figure 1: Example of code change and TODO comment

become incorrect and unreliable documentation of ever-decreasing accuracy, which can mislead developers and cause the introduction of bugs in the future [33, 35]. Therefore, the obsolete TODO comments increase the cost of software maintenance and development, which have negative impact to the quality and reliability of the system. It is thus highly desirable to have a tool that provides just-in-time automatic detection of obsolete TODO comments and removes them before they mislead developers and cause any damages. However, making such a tool is difficult with respect to the following challenges:

- *Capturing comment semantics* – Detecting obsolete TODO comments first requires understanding the semantics of the comments. Compared to source code, comments are written in natural language and have no mandatory format. When source code changes, they can go through a series of software testing to ensure their correctness, however, there is no way to test comments to see if they are still valid or not. Moreover, the source code and comments are of different types (together they form a heterogeneous data) that cannot be easily matched to each other due to their lexical gaps. Therefore, it is a non-trivial task to determine which TODO comments have been addressed and which ones have not.
- *Capturing correlations* – It is very hard to determine if a TODO comment is resolved or not by just reading the source code. A more obvious and reliable way is to check the code change history with respect to the TODO comment, and determine if these code changes resolve the associated task, as shown in Ex.1 in Fig. 1. However, sometimes checking the code changes alone is not sufficient. Ex.2 in Fig. 1 presents such a case. Even though the TODO comment and code change are presented, one can not easily claim this TODO comment is resolved due to his/her unfamiliarity with the code. The associated commit message can provide additional clues to fill this gap. Therefore, to better identify TODO comments are up-to-date or obsolete, it is necessary to consider the correlations between the TODO comments, code changes and commit messages.

In this work, to help developers better maintain the TODO comments in their software systems, we propose a novel neural network model, named **TDCleaner** (**TO**do comment **Cleaner**), which can automatically detect the stale TODO comments in software repositories. TDCleaner consists of two phases: offline learning and online prediction. During offline learning, we collect TODO comments from TOP-10,000 Python and Java Github repositories respectively. We automatically establish positive and negative training samples in terms of whether a TODO comment is resolved or not. Our TDCleaner can be trained as a binary classification model with these two kinds of training samples. To capture the semantics of the heterogeneous data, we employ three encoders, i.e., *TODO Comment Encoder*, *Code Change Encoder*, and *Commit Message Encoder*, to embed TODO comments, code changes, and commit messages into contextualized vectors respectively. TDCleaner then learns correlations and interactions between them by optimizing the final probability score. When it comes to online prediction, for a given TODO comment, we pair it with the associated code change and commit message, and fit them into the trained TDCleaner model to estimate their matching score.

To verify the suitability of our proposed model, we conducted extensive experiments on Python and Java datasets. By comparing with several benchmarks, the superiority of our proposed TDCleaner model is demonstrated. In summary, this work makes the following main contributions:

(1) We propose a novel model, TDCleaner, to automatically detect obsolete TODO comments by mining the histories of the software repositories. TDCleaner can help developers to increase the quality and reliability of software, and alleviate the error-prone code review process.
(2) We build a large dataset for checking obsolete TODO comments from Github repositories, which contains more than 410K TODO comments for Python and more than 350K TODO comments for Java dataset. To the best of our knowledge, this is the first and by far the largest dataset for this task.
(3) We extensively evaluate TDCleaner using real-world popular open-source projects in Github. TDCleaner is shown to outperform several baselines and reduce the developer's efforts in maintaining the TODO comments.
(4) We have released our replication package [37] , including the dataset and the source code of TDCleaner, to facilitate other researchers and practioners to repeat our work and verify their ideas.

The rest of the paper is organized as follows. Section 2 presents the motivating examples and user scenarios of our study. Section 3 presents the details of our approach. Section 4 presents the data preparation for our approach. Section 5 presents the baseline methods , the evaluation metrics, and the evaluation results. Section 6 presents the in-the-wild evaluation. Section 7 presents the threats to validity. Section 8 presents the related work. Section 9 concludes the paper with possible future work.

2 MOTIVATION

We show several motivating examples from popular Github repositories of the sorts of problems mentioned above. We then present user scenarios of employing our proposed approach to address these problems.

2.1 Motivating Examples

Developers change their source code but forget to remove or update associated TODO comments from time to time. Fig 2 shows two

Motivating Example 1: google/clusterfuzz (stars:4.4k)

```
# TODO(ochang): Remove include_strageties once refactor is complete
-  def parse_performance_features(log_lines,
-                                 strategies,
-                                 arguments,
-                                 include_strategies=True):
+  def parse_performance_features(log_lines, strategies, argument):
```

Motivating Example 2: SublimeHaskell/SublimeHaskell (stars:576)

```
class SublimeHaskellAutocomplete(sublime_plugin.EventListener):
    def __init__(self):
        # TODO: Start the InspectorAgent as a separate thread
        self.inspector = InspectorAgent()
-       self.inspector.run()
+       self.inspector.start()
```

Figure 2: Motivating Examples

examples of stale TODO comments we found in real-world Github repositories. We can see that:

Producing obsolete TODO comments is not only done by inexperienced developers: Even developers within experienced teams (such as Ethereum and Google) may ignore or neglect such updates. For example, Motivating Example 1 presents such an obsolete TODO comment example from the *clusterfuzz* project developed by the Google team. The TODO comment within the method `get_client_ip` states that "*remove include_strategies once refactor is complete*". However, when the task was resolved and *include_strategies* was removed from the method, the corresponding TODO comment was not updated. Until the time we report this issue, this stale TODO comment has existed for over 1 year. During this time, such a comment may hinder program comprehension, cause miscommunication between developers, and confuse developers who perform the subsequent development.

A lot of software bugs are caused by the mismatch between code's implementation and developer's intention: While a stale TODO comment itself might be harmless, it can mislead developers and cause the introduction of bugs in the future. For example, in our Motivating Example 2, the task TODO comment was fulfilled and should not be followed any longer. However, since the TODO comment stayed around, other developers can easily misunderstand the software component and violate the assumption and later lead to bugs.

2.2 User Scenarios

We illustrate a usage scenario of TDCleaner as follows:

Without Our Tool: Consider a developer Bob. Daily, when Bob reads other people's code to perform the development, he encounters a few stale TODO comments. These out-of-date TODO comments can clutter the code and have a harmful effect for Bob to understand the current state of the code correctly. Bob may lose confidence in the reliability of the system and even ignore the remainder of useful comments. Furthermore, since Bob has no idea that this TODO comment had already been resolved, Bob tries his best to perform the "pending" task with respect to this task comment by refactoring the code. However, due to his unfamiliarity with the code or the system, these updates are risky and very likely to cause the introduction of bugs in the future.

With Our Tool: Now consider Bob adopts our TDCleaner. Before working on the software, Bob can use our tool to automatically check for the presence of stale TODO comments. Moreover, TDCleaner can identify the specific code change which resolved the TODO comment, thus helping to improve developer's confidence in verifying the obsolete TODO comments. With the help of our tool, Bob successfully removes the stale TODO comments in the current software repository efficiently and accurately, which can increase the reliability and maintenance of the system and decrease the likelihood of introducing bugs.

3 OUR APPROACH

We first define the task of identifying obsolete TODO comments for our study. We then present the details of our proposed model. The overall framework of our approach is illustrated in Fig. 3.

3.1 Task Definition

The motivation of our work is to automatically detect and remove obsolete TODO comments in software projects. To do this, we need to detect if a TODO comment is resolved in a commit or not. We formulate this task as a binary classification learning problem as per below. For a given commit, let **C** be the code changes extracted from the `diff` file, **M** be the corresponding commit message, and **T** be the TODO comment associated with the code changes (if a TODO comment appears within the default number of context lines of a code change, we claim this TODO comment is associated with the code change). Our target is to automatically determine the status **S** of the TODO comment, where **S** is positive when the TODO comment is resolved, and negative when the TODO comment is unresolved. In other words, our goal is to train a model θ using $\langle \mathbf{C}, \mathbf{T}, \mathbf{M} \rangle$ triples such that the probability $P_\theta(\mathbf{S}|\langle \mathbf{C}, \mathbf{T}, \mathbf{M} \rangle)$ is maximized over the given training dataset. Mathematically, our task is defined as finding $\overline{y}$, such that:

$$\overline{y} = argmax_{\mathbf{S}} P_\theta(\mathbf{S}|\langle \mathbf{C}, \mathbf{T}, \mathbf{M} \rangle) \quad (1)$$

$P_\theta(\mathbf{S}|\langle \mathbf{C}, \mathbf{T}, \mathbf{M} \rangle)$ can be seen as the conditional likelihood of predicting the status **S** given the $\langle \mathbf{C}, \mathbf{T}, \mathbf{M} \rangle$ input triples.

3.2 Approach Details

3.2.1 Encoders. Identifying the resolved TODO comments from the unresolved ones is a non-trivial task. This is because the code changes and the TODO comments are of different types (source code vs. natural language) and cannot be easily mapped by simple matching of their lexical tokens. To bridge this gap, TDCleaner adopts three encoders, i.e., *TODO Comment Encoder*, *Code Change Encoder*, and *Commit Message Encoder* to embed the code change, TODO comment and commit message into a vector representation respectively, so that semantically similar concepts across the three modalities can be correlated in the higher dimensional vector space. Through the embedding techniques, heterogeneous data can be easily linked through their vectors.

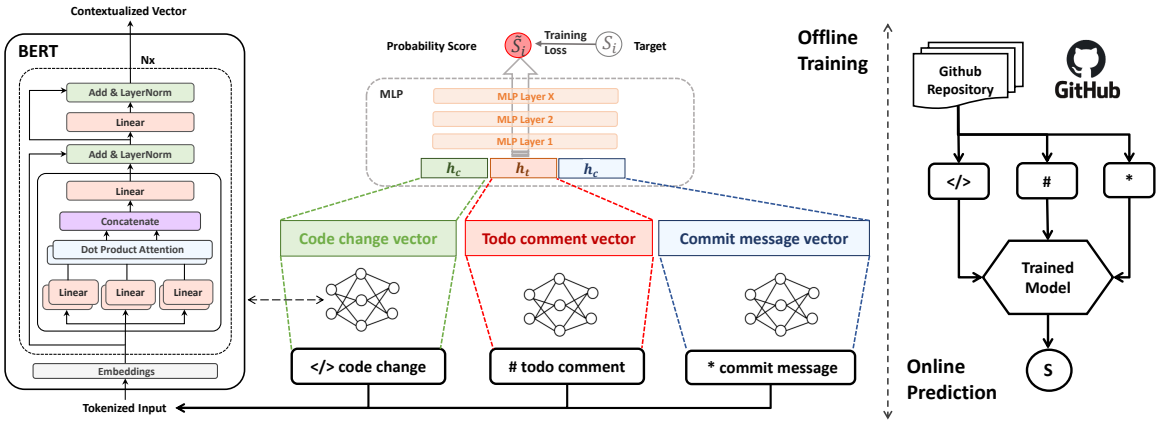

Figure 3: Overview of Our Approach

Recently, neural networks have been widely used to capture the code semantic features by encoding code into vectors [5–7, 9, 12, 19], In this study, we employ BERT [3] as the encoder template for our task. This is because BERT has been proven to be effective for capturing semantics and context information of sentences in many other works [1, 4, 17, 32, 42]. BERT consists of 12-layer transformers [38], each of the transformers being composed of a self-attention sub-layer with multiple attention heads. The input to each BERT embedding component is a sequence of tokens. Given a sequence of tokens $\mathbf{x} = \{x_1, ..., x_T\}$ of length T as input, BERT takes the tokens as input and calculate the contextualized representations $\mathbf{H}^l = \{h_1^l, ..., h_T^l\} \in \mathbb{R}^{T \times D}$ as output, where l denotes the l-th transformer layer and D denotes the dimension of the representation vector. We take the final hidden state of the first special token $\mathbf{h}_1^\mathbf{L}$ as the embedding vector for the input sequence. Our TDCleaner consists of three encoders, i.e., *Code Change Encoder*, *TODO Comment Encoder* and *Commit Message Encoder*. These three encoders are nearly the same in structure and responsible for mapping three kinds of inputs, i.e., code changes, TODO comments and commit messages, into their corresponding embeddings.

- *Code Change Encoder:* The *Code Change Encoder* embeds code changes into vectors. Code changes contain multiple aspects of useful information such as code tokens, added lines and removed lines. Consider a code change $\mathbf{C} = (C_1, C_2, ..., C_{N_C})$ comprising a number of N_C tokens. The *Code Change Encoder* embeds it into a vector $\mathbf{h_c}$ using BERT.
- *TODO Comment Encoder:* The *TODO Comment Encoder* embeds the TODO comment into vectors. Consider a TODO comment $\mathbf{T} = (T_1, T_2, ..., T_{N_T})$ comprising a number of N_T tokens. After feeding $\mathbf{T}$ into the *TODO Comment Encoder*, we can get the embedding vector $\mathbf{h_t}$ for the TODO comment.
- *Commit Message Encoder:* The *Commit Message Encoder* embeds commit messages into vectors. Likewise, the commit message $\mathbf{M} = (M_1, M_2, ..., M_{N_M})$ is embeded into a contextualized vector $\mathbf{h_m}$.

3.2.2 Multi Layer Perceptron. So far, the code change, TODO comment and commit message are represented as independent contextual vectors. To capture the relationships between them, it is necessary to link and fuse their information. To do this, we add a Multi-Layer Perceptron (MLP) [8, 11] to address this need, which is shared with the three encoders. We first concatenate three encoders' contextual vectors, i.e., $\mathbf{h_c}$, $\mathbf{h_t}$, and $\mathbf{h_m}$, to combine the semantic features. To further capture the correlation and reference of the latent feature vectors, we next add a standard MLP on the concatenated vector. In this sense, we can endow the model with a large level of flexibility and non-linearity to learn the interactions between the three encoders. The MLP takes the the contextualized representations (i.e., $\mathbf{h_c}$, $\mathbf{h_t}$, and $\mathbf{h_m}$) as input and outputs the likelihood of the final status $\mathbf{S} = \{0, 1\}$. More precisely, the MLP is defined as follows:

$$
\begin{aligned}
\mathbf{z}_1 &= \phi_1(\mathbf{h}_c, \mathbf{h}_t, \mathbf{h}_m) = \begin{bmatrix} \mathbf{h}_c \\ \mathbf{h}_t \\ \mathbf{h}_m \end{bmatrix} \\
\mathbf{z}_2 &= \phi_2(\mathbf{z}_1) = \mathbf{a}_2(\mathbf{W}_2^T \mathbf{z}_1 + \mathbf{b}_2) \\
&\dots \\
\mathbf{z}_L &= \phi_L(\mathbf{z}_{L-1}) = \mathbf{a}_L(\mathbf{W}_L^T \mathbf{z}_{L-1} + \mathbf{b}_L) \\
P(\mathbf{S} = j | \langle \mathbf{C}, \mathbf{T}, \mathbf{M} \rangle) &= \sigma(\mathbf{z}_L)
\end{aligned}
\tag{2}
$$

$\mathbf{W}_x$, $\mathbf{b}_x$, and $\mathbf{a}_x$ denote the weight matrix, bias vector, and activation function for the x-layer's perceptron respectively. σ is the sigmoid function $\sigma(x) = 1/(1 + e^{-x})$ which will output the final probability of status $\mathbf{S}$ between 0 and 1. For the probability score, we want this score to be high if the TODO comment is resolved and to be low if the TODO comment is unresolved.

4 DATASET PREPARATION

We present the details of our data collection and construction process. We build our dataset by collecting data from the top 10,000 repositories (ordered by the number of stars) in Github for Python and Java repositories. To the best of our knowledge, this is the first and by far the largest dataset for collecting TODO comments from Github repositories.

Commit Message	issue #3326 - adds log for request failing	Issue #372 - Forward all params from client to GitHub
Diff	except requests.exceptions.HTTPError as err: — # TODO: log the message — pass + logging.waring('Fetching comments failed')	# TODO: handle sort and other parapeters — params = {} + params = request.args.copy()
code_change	except requests . exception . httperror as err : <nl> - pass <nl> + logging . warning (' fetching comments failed ')	— params = { } <nl> + params = request . args . copy ( )
commit_msg	issue <issue_id> — adds log for request failing	issue <issue_id> — forward all params from client to github
todo_comment	todo : log the message	todo : handle sort and order parameters
Lebel	*POSITIVE*	*NEGATIVE*

Figure 4: Data Preparation Examples

4.1 Data Collection

4.1.1 Identifying TODO Related Commits. We first clone the top 10,000 repositories from Github. The git repository stores software update history, each update comprises a commit message alongside a `diff` that represents the differences between the current and previous version of the affected files. For each cloned repository, we first checkout all the commits from the repository history. Following that, for each commit, if "TODO" appears within the `diff`, we consider this commit as a *TODO related commit*. We have identified more than 410K TODO related commits from our Python repositories and more than 350K commits from our Java repositories.

4.1.2 Normalize Diff and Commit Message. This step aims to normalize the commit sequence and remove some semantic-irrelevant information. After identifying the TODO related commits, we extract the `diff` and the commit message from the commits and normalize them for later use. For the `diff`, we convert it into lowercase and delete the `diff` header by using regular expressions. Commit IDs within the `diff` are replaced with "⟨commit_id⟩" and commits with a `diff` larger than 1MB are removed. For the commit message, we first lowercase the commit message and retain the first sentences of the *commit message* as the target, since the first sentences are usually the summaries of the entire commit message [10, 15]. Similar to `diff` normalization, Github issue IDs and commit IDs are replaced by "⟨issue_id⟩" and "⟨commit_id⟩" respectively to ensure semantic integrity. The two examples in Fig. 4 demonstrates the process of normalization.

4.1.3 Extract TODO Comments. This step is responsible for extracting TODO comments from the corresponding `diff`. A `diff` may contain multiple TODOs. We remove such instances because they are likely to be comment updates which might introduce noise for the later data construction process. Hereafter, each `diff` contains a single TODO comment and pairs with a commit message. We then use regular expressions to locate comments within the `diff` – if "TODO" appears in the comment, then this comment is identified as a TODO comment. The TODO comment is extracted out of the `diff` as *todo_comment*, and the rest of the `diff` stay the same, referred to as *code_change* in this paper. It is worth mentioning that the difference label, i.e., "+" and "-", are deleted from the TODO comment, otherwise our model can learn directly from the difference labels instead of learning the semantics of the TODO comments. Until now, we are able to establish ⟨*code_change*, *todo_comment*, *commit_msg*⟩ triples.

4.2 Data Construction

4.2.1 Data Labelling. In this work, we consider a `diff` consists of three parts: *LinesAdded*, *LinesRemoved*, and *LinesEqual*. *LinesAdded* are the lines that were added to the current version when compared to its previous version. Similarly, *LinesRemoved* are the lines that were removed from the current version when compared to its previous version. *LinesEqual* were the lines that remained unchanged between two versions. We automatically label each ⟨*code_change*, *todo_comment*, *commit_msg*⟩ triple as follows: (i) if *todo_comment* is within the scope of the *LinesRemoved*, which means the TODO task is already performed by the developer, we label such triple instances as **Positive Samples**. (ii) if *todo_comment* is within the scope of the *LinesEqual*, which means the *code_change* is not responsible for resolving the TODO task, we label such triple instances as **Negative Samples**. (iii) if *todo_comment* is within the scope of the *LinesAdded*, we ignore such triple instances. This is reasonable since the *todo_comment* within the scope of *LinesAdded* corresponds to the first time TODO comments are added. However, our research focuses on detecting obsolete TODO comments from the existing ones. This requires that the TODO comments have been already added to the source code. Fig. 4 demonstrates the data labelling results for a positive sample and a negative sample respectively. We found more than 38K triple instances that are considered as positive samples and 35K triple instances that are considered as negative samples for Python, and 33K positive samples and 32K negative samples for Java.

4.2.2 Manual Checking. We automatically built our positive and negative training samples via heuristic rules. We can not ensure that there are no outlier cases during the label establishment process. Therefore, we performed a manual checking step to examine the label of the training sample is correct or not. We randomly sampled 200 samples (including 100 positive samples and 100 negative samples) from our dataset. Then, the first author manually examined each sample and classified the *todo_comment* is resolved or not based on checking the associated *code_change* and *commit_msg*. Finally, 93 of the positive training samples are marked as resolved and all of the negative training samples are marked as unresolved. Thus, we are confident in the labels of our provided dataset.

4.2.3 Data Splitting. We split the constructed data samples into three chunks: 80 percent of the triple samples are used for training, 10 percent are used for validation and the rest are held-out for testing. The training set is used to adjust the parameters, while the validation set is used to minimize overfitting, and the testing set is used only for testing the final solution to confirm the actual predictive power of our model with optimal parameters. The number of training, validation, test sets of Python and Java dataset are displayed in Table 1.

5 EMPIRICAL EVALUATION

We first present the baselines, the evaluation metrics and our experiment settings. We then describe the results of our quantitative evaluation and manual analysis.

Table 1: Data Statistics

Python	# TODO Commits	416,666
	# Positive samples	38,175
	# Negative samples	35,995
	# Train Set	59,336
	# Val&Test Set	7,417
Java	# TODO Commits	351,266
	# Positive samples	33,797
	# Negative samples	32,490
	# Train Set	53,029
	# Val&Test Set	6,629

5.1 Baselines

To demonstrate the effectiveness of our proposed model, TDCleaner, we compared it with the following chosen baselines:

- **TCO: T**ODO-**C**odeChange-**O**verlap is a reasonable heuristic baseline to identify if a TODO task comment is resolved or not. TCO looks at the overlapping words in the *todo_comment* and *code_change*. For example, as shown in the positive example in 4, the *logging* function was added to the source code to resolve the TODO comment "*TODO: log the message*". Therefore, we created the **TCO** baseline as follows: if there is a common word between the *todo_comment* and the *code_change*, then we declare this TODO task comment to be resolved.
- **TMO: T**ODO-Commit**M**sg-**O**verlap baseline is based on the overlapping words of the *todo_comment* and the *commit_msg*. In general, commit messages are meant to explain the purpose of the source code changes. We thus make the **TMO** baseline as follows: we compare the words in the *todo_comment* and the *commit_msg*, if there is a match we claim this task TODO comment to be resolved.
- **TCMO:** We combined the status predicted by **TCO** and **TMO** to make the **TCMO** baseline. For a testing sample, we declare this task TODO comment as resolved if either **TCO** or **TMO** method predicts so.
- **IRSC:** Sridhara [29] proposed a method, **IRSC** (**I**nformation **R**etrieval Based **S**tatus **C**hecker), that performs well in identifying the status of the task TODO comments. Different from our task, their method requires the signature and body of a method as input, then automatically checks if the associated task comment is up date with the code or obsolete. **IRSC** uses a cosine similarity between the task comment and the *LinesAdded* by incorporating TF-IDF weightings.

5.2 Evaluation Metrics

We define four statistics with respect to our task: (i) True Positive (TP) represents the number of resolved TODO comments that are classified as resolved. (ii) True Negative (TN) represents the number of unresolved TODO comments that are classified as unresolved. (iii) False Positive (FP) represents the number of unresolved TODO comments that are classified as resolved. (iv) False Negative (FN) represents the number of resolved TODO comments that are classified as unsolved. Based on the aforementioned four statistics, we adopted the widely-accepted metrics, i.e., Accuracy, Precision, Recall, and F1-score to evaluate the performance of TDCleaner and baselines.

Our evaluation metrics are defined as follows:

- **Accuracy:** Accuracy represents the proportion of correct predictions (both true positives and true negatives) among the total number of cases examined. The Accuracy metric is defined as follows:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{3}$$

- **Precision:** Precision represents the proportion of TODO task comments that are correctly classified resolved among all the resolved comments. The Precision metric is defined as follows:

$$Precision = \frac{TP}{TP + FP} \tag{4}$$

- **Recall:** Recall represents the proportion of all resolved task comments that are correctly classified as resolved. The Recall metric is defined as follows:

$$Recall = \frac{TP}{TP + FN} \tag{5}$$

- **F1-score:** F1-score is the harmonic mean of precision and recall, which combines both of the two metrics above. It evaluates if an increase in precision (or recall) outweighs a reduction in recall (or precision), respectively. The F1-score metric is defined as follows:

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{6}$$

The higher an evaluation metric, the better a method performs. Note that there is a trade off between Precision and Recall. F1-score provides a balanced view of precision and recall.

5.3 Experimental Settings

We implemented TDCleaner in Python using the Pytorch framework. We used the pre-trained BERT model [3] as the encoder for embedding training samples, which provides a powerful context-dependent sentence representation. BERT can be easily extended to a joint classification model. In our model, *Code Change Encoder*, *TODO Comment Encoder* and the *Commit Message Encoder* are jointly trained to minimize the cross entropy. After the encoding process, *code_change*, *todo_comment* and *commit_msg* will be mapped to a 768 dimensional vector respectively. During the training phase, we optimized the parameters of our model (including the BERT parameters and MLP parameters) using Adam [16] with a batch size of 32. We use the ReLu as the activation function and employ three hidden layers for MLP. A dropout [30] of 0.2 is used for dense layers before computing the final probability. The model is validated every 1,000 batches on the validation set with a batch size of 32. We set the learning rate of Adam to 0.001 and clip the gradients norm by 2. The model with the best performance on the validation set was used for our evaluations.

5.4 Quantitative Analysis

5.4.1 RQ1: The Effectiveness Evaluation. To evaluate the effectiveness of our proposed model, i.e., TDCleaner, we evaluate it and the baseline methods on our testing set in terms of Accuracy, Precision, Recall and F1-score. The evaluation results for Python and Java dataset are shown in Table 2 and Table 3 respectively. From the table, we can observe the following points:

- In general, the methods based on bag-of-words matching, i.e., **TCO** and **TMO**, achieve the worst performance. **TCO** and **TMO** methods identify the resolved TODO comments based on whether common words can be found in the input. As a result, they are unable to consider the context of the *todo_comment*, *code_change*, and *commit_msg* and the relationship between them, which is reflecting that simply checking the overlap words is not enough for our task. It is notable that the **TMO** method gets relatively high Recall, i.e., 75.5% for Python and 73.3% for Java. This is reasonable because *commit_msg* often describes the purpose behind commits, if *commit_msg* matches the *todo_comment*, it is very likely that the developer has completed this task comment and described this update in the *commit_msg*, an example is shown in the positive example of Fig. 4. However, not all commit messages are meaningful and related to the task comment. This also explains its surprisingly low score on Precision and F1 score.
- **TCMO** performs better than **TCO** and **TMO** respectively. For example, it improves over **TCO** on F1-score by 12.8% on Python dataset and Java dataset, and it improves over **TMO** on F1-score by 62.1% on Python dataset and 66.8% on Java dataset. Instead of solely based on *code_change* or *commit_msg*, **TCMO** combines the advantage of **TCO** and **TMO** by incorporating useful information in both *code_change* and *commit_msg*. This results in its superior performance to the other two approaches. It also signals that *code_change* and *commit_msg* convey much valuable information for our task of identifying the obsolete TODO comment.
- **IRSC** has its advantage as compared to the words overlapping based methods, i.e., **TCO**, **TMO** and **TCMO**. Rather than checking if the words are overlapping, **IRSC** employed the TF-IDF metric, which can extract descriptive terms and identify the up-to-date status of the TODO comment by computing their similarities. However, TF-IDF is still based on bag-of-words model, therefore **IRSC** can only capture the lexical level features, but unable to capture the semantic features and co-occurrences in separate input sequences. That is why its performance is comparatively suboptimal.
- **Our new model, i.e.,** TDCleaner**, outperforms all the baseline methods by a large margin** in terms of all evaluation metrics. We attribute this to the following reasons: First, it uses ⟨*code_change*, *todo_comment*, *commit_msg*⟩ triple as input which considers the useful information across different resources. Second, compared with bag-of-words models, the advantage of our proposed model is also clear. Our model employs **BERT** as encoders to embed *code_change*, *todo_comment*, *commit_msg* into high dimensional vectors. These vector representations learn the semantic and structural features from the input and assists TDCleaner to separate resolved task comments from the unresolved ones.
- By comparing the evaluation results of the different datasets, i.e., Python and Java, we can see that TDCleaner is stably and substantially better than the other baselines. This suggests that our approach behaves consistently across different programming languages. This supports the likely generalization and robustness of our approach, which also justifies that our approach is language-independent and we believe it can be easily adapted to other programming languages.

Table 2: Effectiveness Evaluation (Python)

Measure	Accuracy	Precision	Recall	F1
TCO	55.3%	45.3%	58.1%	50.9%
TMO	56.9%	23.1%	75.5%	35.4%
TCMO	58.4%	54.8%	60.2%	57.4%
IRSC	60.4%	61.0%	62.1%	61.6%
TDCleaner	**84.7%**	**82.6%**	**86.8%**	**84.7%**

Table 3: Effectiveness Evaluation (Java)

Measure	Accuracy	Precision	Recall	F1
TCO	56.5%	47.5%	58.7%	52.5%
TMO	56.9%	23.4%	73.3%	35.5%
TCMO	59.7%	57.6%	60.8%	59.2%
IRSC	60.1%	59.0%	70.1%	64.0%
TDCleaner	**85.0%**	**86.2%**	**84.4%**	**85.3%**

Answer to RQ-1: How effective is our approach for identifying the obsolete TODO comments? – we conclude that our approach is highly effective for identifying the resolved TODO comments from the unresolved ones.

5.4.2 RQ2: Component-Wise Evaluation. The key to our obsolete TODO detection task is to effectively capture the relationship and references between code changes and TODO comments. To do so, we adopt three encoders, i.e., *Code Change Encoder*, *TODO Comment Encoder*, and *Commit Mesage Encoder* to better represent and link information between *code_change*, *todo_comment* and *commit_msg*. To verify the effectiveness of these three encoders, we conduct a component-wise evaluation to evaluate their individual performance as well as their contributions one by one. We compare TDCleaner with three of its incomplete versions:

(1) **TD_CC_Encoder:** it keeps the *TODO Comment Encoder* and the *Code Change Encoder*. It does not consider *Commit Message Encoder* for this model. It is then trained as a binary classification model by using ⟨*code_change*, *todo_comment*⟩ pairs as the input.

(2) **TD_MSG_Encoder:** it keeps the *TODO Comment Encoder* and the *Commit Message Encoder*. It does not consider *Code*

Table 4: Component-Wise Evaluation (Python)

Measure	Accuracy	Precision	Recall	F1
TD_CC_Encoder	78.4%	78.5%	79.0%	78.8%
TD_MSG_Encoder	61.9%	62.0%	63.0%	62.5%
CC_MSG_Encoder	76.1%	67.7%	82.4%	74.3%
TDCleaner	**84.7%**	**82.6%**	**86.8%**	**84.7%**

Table 5: Component-Wise Evaluation (Java)

Measure	Accuracy	Precision	Recall	F1
TD_CC_Encoder	78.6%	82.6%	76.9%	79.7%
TD_MSG_Encoder	60.9%	59.6%	61.8%	60.7%
CC_MSG_Encoder	77.0%	81.3%	75.3%	78.2%
TDCleaner	**85.0%**	**86.2%**	**84.4%**	**85.3%**

Change Encoder for this model. **TD_MSG_Encoder** can be trained with ⟨*todo_comment*, *commit_msg*⟩ pairs same as above.

(3) **CC_MSG_Encoder:** it keeps the *Code Change Encoder* and *Commit Message Encoder*. It ignores the *TODO Comment Encoder* for this model. Likewise, **CC_MSG_Encoder** is trained with *code_change* and *commit_message* as input the same with above.

(4) TDCleaner**:** our model which contains all three Encoders.

The evaluation results are shown in Table 4 and Table 5 for Python and Java respectively. It can be seen that:

- **No matter which component we removed, it hurts the overall performance of our model.** This verifies our assumption that all the three encoders embed useful information from their input respectively.
- The performance of **TD_CC_Encoder is better than the other two variants**. In other words, keeping the *TODO Comment Encoder* and the *Code Change Encoder* achieve a minimal performance drop. This justifies the importance and necessity of the above two encoders.
- **TD_MSG_Encoder achieves the worst performance.** It is clear there is a significant drop overall in every evaluation measure after removing the *Code Change Encoder*. This signals that the *Code Change Encoder* is the most important of all the three encoders and has major contributions to the overall performance.
- **Even though TD_CC_Encoder don't get top results as** TDCleaner**, it still achieves considerable performance** (e.g., with F1 score close to 80%), which further confirms the strength of our approach. Moreover, since **TD_CC_Encoder** does not rely on the commit messages, it can be used as a light variant of our approach, which can help developers to remove the obsolete TODO comment just-in-time when code change happens.

Answer to RQ-2: How effective is our use of *TODO Comment Encoder, Code Change Encoder* and *Commit Message Encoder* under automatic evaluation? – we conclude that all the three encoders are effective and helpful to enhance the performance of our model.

Table 6: Ablation Evaluation

Measure	Python		Java	
	Drop-BERT	**Ours**	**Drop-BERT**	**Ours**
Accuracy	64.6%	**84.7%**	66.2%	**85.0%**
Precision	65.2%	**86.2%**	65.2%	**86.2%**
Recall	65.9%	**86.8%**	71.1%	**84.4%**
F1	65.6%	**84.7%**	68.1%	**85.3%**

5.4.3 RQ3: Ablation Evaluation. Another advantage of TDCleaner is using a pre-trained BERT model to learn semantic features from the input.The BERT model provides a powerful context-dependent sentence representation and has achieved the state-of-the-art performance on various NLP tasks, i.e., question answering, language inference, etc. Therefore, we conduct an ablation analysis to verify the effectiveness of using BERT as embedding techniques for our task. In this research question, we compare our proposed model with **Drop-BERT**. **Drop-BERT** drops BERT from our model and does not use the embeddings pre-trained by BERT. Instead we replaced them with a traditional Word2Vec embedding technique.

The results of our comparisons are presented in Table 6. By comparing the results of **Drop-BERT** and **Ours**, we can measure the performance improvement achieved by employing the BERT embedding techniques. **It is clear that dropping the BERT component hurts the performance of our model**. For example, regarding the F1 score, the improvements achieved by adding BERT embeddings range from 25.2% to 29.1%. These results indicate that the BERT embeddings are useful and effective for our task of identifying the obsolete TODO comments.

Answer to RQ-3: How effective is our approach for using BERT embedding techniques? – we conclude that BERT model significantly improves the overall performance of our model.

5.5 Manual Analysis

To better understand the reasons why TDCleaner outperforms other approaches, we manually inspected the test results. Based on our inspection, we summarize two major advantages of TDCleaner as compared to other baseline approaches:

First, TDCleaner automatically learns important semantic-level features from the vector representations, while the bag-of-words based models are limited to learn the lexical-level features. All of the baseline approaches, i.e., **TCO**, **TMO**, **TCMO** and **IRSC** rely on the lexical similarities between the input sequences, they are unable to consider semantic-level features. In contrast, our TDCleaner leverages a probabilistic model to learn semantic features and capture common patterns from a very large scale of ⟨*code_change*, *todo_comment*, *commit_msg*⟩ training samples. These patterns learned by TDCleaner can cover more and diverse samples. For example, as shown in the test sample 2 of Fig. 5, there is no similarity among words between the *todo_comment* and *code_change* or *commit_msg*, thus **TCO**, **TMO** and **TCMO** fail to output the correct prediction, while TDCleaner successfully learns the semantics behind the TODO comment "*TODO - restore this*", and therefore TDCleaner detects this task is completed. Moreover, relying on words

Test sample 1: (Negative)

```
# TODO: not sure what to put for message_id.
yield from self._submit_request_to_server(
-   message_id=share.job_id,
-   job_id=share.job_id,
+   message_id=share.work_id,
+   work_id=share.work_id,
```

Commit Message: changed all the references to job_id to work_id

TCO ✗ TMO ✓ TCMO ✗ IRSC ✗ TDCleaner ✓

Test sample 2: (Positive)

```
- / / TODO - restore this:
- / / validateCons1(new int[]{1}, new int[]{4}, Roi.TRACED_ROI)
+ validateCons1(new int[]{1}, new int[]{4}, Roi.TRACED_ROI)
```

Commit Message: Finish drawPixels() test

TCO ✗ TMO ✗ TCMO ✗ IRSC ✗ TDCleaner ✓

Test sample 3: (Positive)

```
-   # TODO: This action (and related code) will be deprecated
-   self.__create_keywords_action()
```

Commit Message: remove deprecated code

TCO ✗ TMO ✗ TCMO ✗ IRSC ✗ TDCleaner ✓

Figure 5: Manual Analysis Examples

Summary
Remove the obsolete TODO comment

ISSUE TYPE
• Bugfix Pull Request

COMPONENT NAME
scripts/recipe_robot_lib/recipe_generator.py (line:1702)

ADDITIONAL INFORMATION
The pending task of the TODO comment has already performed in the earlier version, but the TODO comment was not removed accordingly the moment. Please check the following commit:
a47f2e6

Figure 6: Pull Request Examples

overlapping leads to unreliable and inaccurate results, as shown in test sample 1, the developer replaced *job_id* with *work_id*, however since the word "*message_id*" appeared in both *todo_comment* and *code_change*, **TCO**, **TCMO** and **IRSC** falsely identifies that this comment has been resolved.

Second, TDCleaner learns the correlations and references between the *code_change*, *todo_comment* and *commit_msg*. All the baselines do not take the co-evolution between code change and comment updates into consideration, and thus can not detect the interaction matching between them. Different from the baseline methods, TDCleaner explicitly adopts three encoders and the MLP layer between them, which enables our model to effectively link the interaction relationships between *todo_comment* with *code_change* and *commit_msg*. Fig. 5 test sample 3 presents such an example, all the baseline approach fail to handle such a case, however, our approach relates the task comment ("*TODO: This action (and related code) will be deprecated*") with the removed code lines and the commit message ("*remove deprecated code*"), thus successfully identifies this TODO comment as resolved.

We also inspected a number of samples where TDCleaner failed to make correct predictions. These samples presented two common aspects that TDCleaner is difficult to deal with. A common failed situation is that the code change or commit message do not provide sufficient information to judge the status of the TODO comment. For example, in a test sample, the developer removed the comment "TODO: is the full url right?" after adding a print statement "print res.data". However, since we lack contextual information of the data structure, TDCleaner is not able to perform such a classification. Another situation is that the code changes are too complicated for TDCleaner to learn. For example, to implement the TODO task, developers sometimes make many code changes across different sub modules. The complicated code changes can hinder our approach for capturing the code semantics. We will explain such examples in the camera ready paper and note them as opportunities for future research.

Why Does Our Approach Succeed? – we conclude that our approach automatically learns the correct semantics and correlations from the training samples.

6 IN-THE-WILD EVALUATION

Since the final goal of our approach is detecting and removing the obsolete TODO comments in software projects, we also perform an in-the-wild evaluation to evaluate the effectiveness of TDCleaner for removing the stale TODO comments in real world software repositories in Github.

We randomly selected 100 Python repositories hosted on Github; for each Github repository, we first clone the repository from Github to local. Following that, we checkout all the commits of the repository to identify the TODO related commits. By going through the same steps of normalization, we extract the *todo_comment*, *code_change* and *commit_msg* from `diff` and commit messages. We then build ⟨*code_change*, *todo_comment*, *commit_msg*⟩ triple and feed the triple as input for TDCleaner. For each constructed triple input, we apply TDCleaner to check if the TODO comment has been resolved or not. If the TODO comment has been identified as resolved by our approach but not removed accordingly, we regard this TODO comment as obsolete.

We ran TDCleaner on these software projects and TDCleaner reported that 23 TODO comments as being obsolete. We manually checked all of these reported TODO comments, 5 of them are "intermediate" obsolete TODO comments, which means they have been removed in subsequent versions. The reason for this phenomenon maybe that developers often clean up the obsolete TODO comments before specific software releases. Even though these TODO

Repo: biocode

```
# TODO: Parse description lines from within each script
-   json_data[category].append({'name': script, 'desc': ''})
+   json_data[category].append({'name': script, 'desc': desc})
```

Repo: tea-lang

```
# TODO: Check that var is ordinal. If so, then assign all ordinal values numbers
-   data.append(var_data)
+   if var.is_ordinal():
+       ordered_cat = var.metadata[categories]
+       num_var_data = [ordered_cat[v] for v in var_data]
+       var_data = num_var_data
```

Repo: carefree-learn

```
# TODO: utilize cv_weights with sample_weights[split.split_indices]
  if sample_weights is not None:
      self.tr_weights = sample_weights[split.remained_indices]
+     self.cv_weights = sample_weights[split.split_indices]
```

Figure 7: Practical Analysis Examples

comments are finally removed during software development before being cleaned up by developers, they may mislead developers, waste their time to check the implementations, complicate the code review and cause the introduction of bugs. It is necessary to remove the obsolete TODO comments just-in-time or avoid introducing them at all.

In addition to the 5 "intermediate" obsolete TODO comments, there are still 18 "potential" obsolete TODO comments dangling in the current version of these projects. For each of these "potential" obsolete TODO comments, we would like to investigate if it is really an obsolete comment based on developer feedback. To do this we submitted a pull-request to the corresponding Github repository by removing this "potential" obsolete TODO comment. To help developers verify the pull-request more easily and confidently, we also notified the developers for the commit for which our approach identifies this TODO comment has been resolved. An example of our submitted pull-request is shown in 7. To avoid subjective bias, the developers don't know the pull-requests are automatically detected by our tool TDCleaner. We submitted 18 pull-requests and 9 of them have already been confirmed and merged by the developers. 4 of them were closed by developers and the rest of them are still open. Fig. 7 demonstrates three examples we submitted to the developers. From developers feedback to us we make the following observations:

- For developers who have adopted our pull-request changes, they appreciated the contributions we made to their repositories. This signals that our proposed model, i.e., TDCleaner, has positive effects to the robustness of the system and may decrease the cost of its development and maintenance.
- Some developers expressed their curiosity about how we detected these obsolete TODO comments. For example, when we submitted pull-request to the first repository in Fig. 7, a developer mentioned, *"Indeed, thank you! I wonder how did you find this? Just browsing the code at leisure time? :)"*. This is because this Github repository contains more than 4,000 commits, it is thus very hard, if not possible, to manually identify an obsolete TODO comment from the huge amount of commits accumulated in the repository. This also justifies the importance and necessity of developing a tool to automatically detect the obsolete TODO comments.
- Instead of completely removing the obsolete TODO comments, some developers suggested another way of just removing the word "TODO". The second example in Fig 7 presents such a situation. When we submitted the pull-request to remove the obsolete TODO comment, the developer responded *"How about just removing the word "TODO"? The rest of the comment is now good documentation."*. This may be a reasonable approach because the TODO comments often describe the pending tasks that should be done, once the developer completed the task, the rest of the TODO could potentially serve as good documentations for the newly added code.
- Not all the pull-requests we submitted are accepted by developers. The third case in Fig 7 shows such an example, the developer closed our pull-request and commented *"Good catch! However although I've already recorded cv_weights, I haven't actually utilized it yet. Which means, I haven't applied cv_weights to get a weighted metrics on the validation set."* This indicates that sometimes the information is not enough for TDCleaner to infer the correct prediction, for such cases, we still need developers to double check the results.

How effective is our TDCleaner for removing obsolete TODO comments in real Github repositories? – we conclude that our model is effective for detecting obsolete TODO comments in real word Github repos.

7 THREATS TO VALIDITY

Several threats to validity are related to our research: In the data preparation process, we removed the diffs which contain multiple TODOs. When we evaluate our approach in practice, the evaluation data goes through the same data processing steps, which means the diffs with multiple TODOs are also removed before feeding them into our model. We have to admit our current approach lacks the ability of dealing with multiple TODOs when analyzing unseen data. We will try to address this shortcoming in our future work.

Besides, our dataset is constructed from Python and Java projects in Github. This was because Python and Java are the most popular programming languages widely used by developers in GitHub. However, there are many other programming language projects in Github which were not considered in our study. Considering that our approach is language-independent, we argue that our approach can be easily adapted to other programming languages. We will try to extend our work to benefit more developers in the future.

Regarding the model hyperparameters, there are two key hyperparameters for constructing our model, i.e., the embedding size of the encoders and the size of MLP hidden layers. Because we use the pre-trained BERT model as our encoders, the embedding size is fixed to 768 for BERT. Therefore, the only hyperparameter

we can tune is the size of the hidden layers. Theoretically, we can fine-tune the size of hidden layers, however, due to the wide range of different size settings, the complexity of training is too expensive and time-consuming. We thus follow classic MLP layer settings [8] in previous works and our approach has achieved promising results for the task.

Another threat to validity is that our work is focused on TODO comment in particular, which is a special case of Self-Admitted Technical Debt (SATD) [26]. Our preliminary study focuses on TODOs as opposed to other SATD, such as TODO, HACK, FIXME, etc. This is because a TODO often hints at the functionality that is not yet implemented, while HACK and FIXME might point to existing, but imperfect implementations. Therefore, we choose the obsolete TODO comments for our preliminary study, we plan to investigate the effectiveness of our approach to other types of STAD comments in the future.

8 RELATED WORK

8.1 TODO Comments in SE Research

TODO comments are widely used by developers to notify themselves or others of pending tasks. Prior works have investigated the role of TODO comments in software development and maintenance [13, 23, 26, 28, 29, 31, 40].

For example, Storey et al. [31] performed an empirical study among developers to investigate how TODO comments are used and managed in software engineering. They found that the use of task annotations varies from individuals to teams, and if incorrectly managed, they could negatively impact the maintenance of the system. Potdar and Shihab [26] proposed the self-admitted technical debt (SATD) concept (e.g., TODO, FIXME, HACK) for the first time, and found that 26.3-63.5% of the SATD was removed after its introduction. Huang et al. [13] used the text-mining based methods to predict whether a comment contains SATD or not. Rungroj et al. [21] first introduced the concept of "on-hold" SATD and proposed a tool [20] to identify and remove the "on-hold" SATD automatically. Even if obsolete TODOs and "on-hold" SATD are similar, there are several differences between our work and theirs: (i) According to their definition in [21], the "on-hold" SATD contains a specific condition to be triggered, while the obsolete TODOs in our study do not have such constraint. (ii) their approach [20] detects "on-hold" SATD based on the issue-referring comments, while our approach identifies general obsolete TODO comments based on commit histories.

There is very limited work on detecting obsolete TODO comments. Sridhara [29] presented a technique for automatically identifying whether a TODO comment in a given method is up to date or obsolete. Different from their approach which requires the method's body and signature as input, our TDCleaner takes the commits as input which is more general. Moreover, their approach is based on lexical level and heuristic rules defined by human experts, which can not capture the semantic features and/or adapt to different kinds of samples. We compared their approach, i.e., **IRSC** with TCleaner in Section 5.4, the evaluation results show that our approach outperforms theirs by a large margin.

8.2 Code-Comment Inconsistency Detection

Inconsistent updates between code and comments are risky and should be carefully reviewed by practitioners. Previous studies [18, 19, 22, 24, 27, 33–36, 43] have investigated the inconsistency between code and their documentations.

Some works focused on the comments related to specific code properties or of specific types. For example, Tan et al. [33–35] proposed a series of approaches to detect code-comment inconsistency with respect to specific code properties, such as lock mechanisms, function calls and interrupts. They use NLP techniques to extract the concept-related rules and use static program analysis to check source code against these rules. Some other works [18, 22, 27] focused general comments and took code change into consideration. For example, Liu et al. [18] leveraged 64 manually-crafted features and machine learning techniques to check the code comment inconsistency. Most recently, Liu et al. [19] employed a seq2seq model to automatically update bad comments in software projects. The techniques mentioned above check the inconsistency between code and comments. This is different from our work which is designed to detect obsolete TODO comments in software repositories.

9 CONCLUSION AND FUTURE WORK

This research aims to automatically detect and remove obsolete TODO comments from software repositories. To address this task, we first collect obsolete TODO comments from the top-10,000 Github repositories, To the best of our knowledge, this is the first and by far the largest dataset for detecting obsolete TODO comments. We propose an approach named TDCleaner (**TODO comment Cleaner**), which leverage a neural network model to learn the semantic features and correlations between code changes, TODO comments and commit messages. Extensive experiments on the real-world Github repositories have demonstrated its effectiveness and promising performance. In the future, we plan to investigate the effectiveness of TDCleaner with respect to other programming languages. We also plan to adapt TDCleaner to other types of task comments, such as FIXME, HACK, etc.

ACKNOWLEDGMENT

This research was partially supported by the ARC Laureate Fellowship FL190100035, and the National Research Foundation, Singapore under its Industry Alignment Fund – Prepositioning (IAF-PP) Funding Initiative. Any opinions, findings, and conclusions, or recommendations expressed in this material are those of the author(s) and do not reflect the views of the National Research Foundation, Singapore. The authors would like to thank the reviewers for the insightful and constructive feedback.

REFERENCES

[1] Iz Beltagy, Kyle Lo, and Arman Cohan. 2019. SciBERT: A pretrained language model for scientific text. *arXiv preprint arXiv:1903.10676* (2019).
[2] Sergio Cozzetti B de Souza, Nicolas Anquetil, and Káthia M de Oliveira. 2005. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*. 68–75.
[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).
[4] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained

model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[5] Zhipeng Gao, Vinoj Jayasundara, Lingxiao Jiang, Xin Xia, David Lo, and John Grundy. 2019. Smartembed: A tool for clone and bug detection in smart contracts through structural code embedding. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 394–397.

[6] Zhipeng Gao, Lingxiao Jiang, Xin Xia, David Lo, and John Grundy. 2020. Checking smart contracts with structural code embedding. *IEEE Transactions on Software Engineering* (2020).

[7] Zhipeng Gao, Xin Xia, John Grundy, David Lo, and Yuan-Fang Li. 2020. Generating question titles for stack overflow from mined code snippets. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 4 (2020), 1–37.

[8] Zhipeng Gao, Xin Xia, David Lo, and John Grundy. 2020. Technical Q8A Site Answer Recommendation via Question Boosting. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 1 (2020), 1–34.

[9] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 933–944.

[10] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 631–642.

[11] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. 2017. Neural collaborative filtering. In *Proceedings of the 26th international conference on world wide web*. 173–182.

[12] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*. ACM, 200–210.

[13] Qiao Huang, Emad Shihab, Xin Xia, David Lo, and Shanping Li. 2018. Identifying self-admitted technical debt in open source projects using text mining. *Empirical Software Engineering* 23, 1 (2018), 418–451.

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

[15] Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically generating commit messages from diffs using neural machine translation. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 135–146.

[16] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[17] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. 2019. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461* (2019).

[18] Zhiyong Liu, Huanchao Chen, Xiangping Chen, Xiaonan Luo, and Fan Zhou. 2018. Automatic detection of outdated comments during code changes. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. IEEE, 154–163.

[19] Zhongxin Liu, Xin Xia, Meng Yan, and Shanping Li. 2020. Automating Just-In-Time Comment Updating. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 585–597.

[20] Rungroj Maipradit, Bin Lin, Csaba Nagy, Gabriele Bavota, Michele Lanza, Hideaki Hata, and Kenichi Matsumoto. 2020. Automated Identification of On-hold Self-admitted Technical Debt. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 54–64.

[21] Rungroj Maipradit, Christoph Treude, Hideaki Hata, and Kenichi Matsumoto. 2020. Wait for it: identifying "On-Hold" self-admitted technical debt. *Empirical Software Engineering* 25, 5 (2020), 3770–3798.

[22] Haroon Malik, Istehad Chowdhury, Hsiao-Ming Tsou, Zhen Ming Jiang, and Ahmed E Hassan. 2008. Understanding the rationale for updating a function's comment. In *2008 IEEE International Conference on Software Maintenance*. IEEE, 167–176.

[23] Pengyu Nie, Rishabh Rai, Junyi Jessy Li, Sarfraz Khurshid, Raymond J Mooney, and Milos Gligoric. 2019. A framework for writing trigger-action todo comments in executable format. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 385–396.

[24] Sheena Panthaplackel, Pengyu Nie, Milos Gligoric, Junyi Jessy Li, and Raymond J Mooney. 2020. Learning to update natural language comments based on code changes. *arXiv preprint arXiv:2004.12169* (2020).

[25] David Lorge Parnas. 2011. Precise documentation: The key to better software. In *The Future of Software Engineering*. Springer, 125–148.

[26] Aniket Potdar and Emad Shihab. 2014. An exploratory study on self-admitted technical debt. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 91–100.

[27] Inderjot Kaur Ratol and Martin P Robillard. 2017. Detecting fragile comments. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 112–122.

[28] Xiaoxue Ren, Zhenchang Xing, Xin Xia, David Lo, Xinyu Wang, and John Grundy. 2019. Neural network-based detection of self-admitted technical debt: From performance to explainability. *ACM transactions on software engineering and methodology (TOSEM)* 28, 3 (2019), 1–45.

[29] Giriprasad Sridhara. 2016. Automatically detecting the up-to-date status of ToDo comments in Java programs. In *Proceedings of the 9th India Software Engineering Conference*. 16–25.

[30] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research* 15, 1 (2014), 1929–1958.

[31] Margaret-Anne Storey, Jody Ryall, R Ian Bull, Del Myers, and Janice Singer. 2008. TODO or to bug. In *2008 ACM/IEEE 30th International Conference on Software Engineering*. IEEE, 251–260.

[32] Jeniya Tabassum, Mounica Maddela, Wei Xu, and Alan Ritter. 2020. Code and named entity recognition in stackoverflow. *arXiv preprint arXiv:2005.01634* (2020).

[33] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /* iComment: Bugs or bad comments?*. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. 145–158.

[34] Lin Tan, Ding Yuan, and Yuanyuan Zhou. 2007. Hotcomments: how to make program comments more useful?. In *HotOS*.

[35] Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. 2011. aComment: mining annotations from comments and code to detect interrupt related concurrency bugs. In *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 11–20.

[36] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T Leavens. 2012. @ tcomment: Testing javadoc comments to detect comment-code inconsistencies. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 260–269.

[37] TDCleaner. 2020. *Our replicate package.* https://github.com/beyondacm/TDClearner

[38] 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*. 5998–6008.

[39] Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. 2019. A large-scale empirical study on code-comment inconsistencies. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 53–64.

[40] Meng Yan, Xin Xia, Emad Shihab, David Lo, Jianwei Yin, and Xiaohu Yang. 2018. Automating change-level self-admitted technical debt determination. *IEEE Transactions on Software Engineering* 45, 12 (2018), 1211–1229.

[41] Annie TT Ying, James L Wright, and Steven Abrams. 2005. Source code that talks: an exploration of Eclipse task comments and their implication to repository mining. *ACM SIGSOFT software engineering notes* 30, 4 (2005), 1–5.

[42] Ting Zhang, Bowen Xu, Ferdian Thung, Stefanus Agus Haryono, David Lo, and Lingxiao Jiang. 2020. Sentiment Analysis for Software Engineering: How Far Can Pre-trained Transformer Models Go?. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 70–80.

[43] Hao Zhong and Zhendong Su. 2013. Detecting API documentation errors. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*. 803–816.