Just-In-Time TODO-Missed Commits Detection

Haoye Wang, Zhipeng Gao, Xing Hu, David Lo, John Grundy and Xinyu Wang

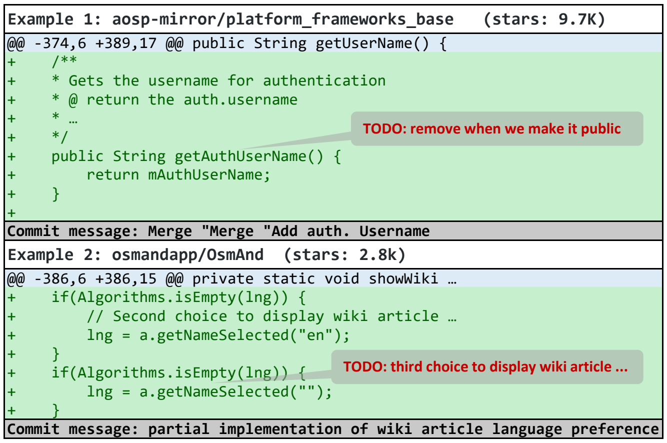
Abstract—TODO comments play an important role in helping developers to manage their tasks and communicate with other team members. TODO comments are often introduced by developers as a type of technical debt, such as a reminder to add/remove features or a request to optimize the code implementations. These can all be considered as notifications for developers to revisit regarding the current suboptimal solutions. TODO comments often bring short-term benefits – higher productivity or shorter development cost – and indicate attention needs to be paid for the long-term software quality. Unfortunately, due to their lack of knowledge or experience and/or the time constraints, developers sometimes may forget or even not be aware of suboptimal implementations. The loss of the TODO comments for these suboptimal solutions may hurt the software quality and reliability in the long-term. Therefore it is beneficial to remind the developers of the suboptimal solutions whenever they change the code. In this work, we refer this problem to the task of detecting *TODO-missed commits*, and we propose a novel approach named TDREMINDER (**T**O**D**O comment **Reminder**) to address the task. With the help of TDREMINDER, developers can identify possible missing TODO commits just-in-time when submitting a commit. Our approach has two phases: offline training and online inference. We first embed code change and commit message into contextual vector representations using two neural encoders respectively. The association between these representations is learned by our model automatically.In the online inference phase, TDREMINDER leverages the trained model to compute the likelihood of a commit being a *TODO-missed commit*. We evaluate TDREMINDER on datasets crawled from 10k popular Python and Java repositories in GitHub respectively. Our experimental results show that TDREMINDER outperforms a set of benchmarks by a large margin in *TODO-missed commits* detection. Moreover, to better help developers use TDREMINDER in practice, we have incorporated Large Language Models (LLMs) with our approach to provide explainable recommendations. The user study shows that our tool can effectively inform developers not only "when" to add TODOs, but also "where" and "what" TODOs should be added, verifying the value of our tool in practical application.

Index Terms—Technical Debt, TODO Comment, Code-Comment Inconsistency, Suboptimal Implementation

1 INTRODUCTION

Natural language annotations are an important part of software repositories and are used to communicate between developers [1], [2]. Terms such as ”FIXME”, ”XXX”, and ”TODO” are frequently employed to indicate instances of technical debt (TD) or suboptimal implementations that demand future attention [3], [4], [5]. Within these comments, TODO comments are the most extensively used by developers to describe pending tasks during the software development lifecycle. For example, TODO comments can be used as a reminder to add/delete features, or a request for other members to look at a problem, or a request to optimize/clean/refactor the source code. A previous study by Ying et al. [6] identified the importance and frequency of TODO comments in software repositories.

Different from some previous work in the field of self-admitted technical debt (SATD) [7], [8], [9], [10], we explore the TODO comments from a constructive viewpoint. We suggest that the presence of TODO comments can be beneficial. Such explicit markers by developers enhance the chances of addressing these suboptimal sections in future iterations. TODO comments usually contain information about code changes that can improve software quality, performance, maintenance, and reliability [11]. In other words, TODO comments indicate the existence of (temporary) suboptimal solutions to achieve short-term goals, such as higher productivity, satisfying testing needs, or meeting release time constraints. As an example, consider Example 1 in Figure 1. Here, the developer implemented the `getAuthUserName` function for short-term benefits. This code implementation contains a potential risk that the authentication user name may leak out when it is publicly



```
Example 1: aosp-mirror/platform_frameworks_base   (stars: 9.7K)
@@ -374,6 +389,17 @@ public String getUserName() {
+    /**
+    * Gets the username for authentication
+    * @ return the auth.username
+    * …
+    */
+    public String getAuthUserName() {
+        return mAuthUserName;
+    }
+
Commit message: Merge "Merge "Add auth. Username
Example 2: osmandapp/OsmAnd  (stars: 2.8k)
@@ -386,6 +386,15 @@ private static void showWiki …
+    if(Algorithms.isEmpty(lng)) {
+        // Second choice to display wiki article …
+        lng = a.getNameSelected("en");
+    }
+    if(Algorithms.isEmpty(lng)) {
+        lng = a.getNameSelected("");
+    }
Commit message: partial implementation of wiki article language preference
```

TODO: remove when we make it public
TODO: third choice to display wiki article ...

Fig. 1. Examples of TODO comments

- *Haoye Wang is with Hangzhou City University, China.*
  E-mail: wanghaoye@hzcu.edu.cn
- *Zhipeng Gao is with Shanghai Institute for Advanced Study of Zhejiang University, China.*
  E-mail: zhipeng.gao@zju.edu.cn
- *Xing Hu and Xinyu Wang are with Zhejiang University, China.*
  E-mail: {xinghu, wangxinyu}@zju.edu.cn
- *David Lo is with Singapore Management University, Singapore.*
  E-mail: davidlo@smu.edu.sg
- *John Grundy is with Monash University, Melbourne, Australia.*
  E-mail: john.grundy@monash.edu
- *Xinyu Wang is the corresponding author.*

accessible. This issue could hurt software quality and cause software security problems. However, due to the code being unremarkable or the developers' lack of familiarity or development experience, they may be completely unaware of the current suboptimal code changes, e.g., when taking over responsibility for the code from another developer. If the TODO comment in the figure could be promptly added, then this risk would gain more attention from developers. Therefore, it is helpful and beneficial to prompt the developers to insert notes of their suboptimal code (as comments) before submitting a commit.

In this paper, to address such issues, we propose an automated solution that can detect whether a submitted commit is a *TODO-missed commit*. We define a commit as a *"TODO-missed commit"* if its associated change is *suboptimal* and there is a need to introduce further TODO actions. These *TODO-missed commits* have negative impact to the quality and reliability of software. Once a *TODO-missed commit* is identified, we can provide a just-in-time reminder to the developer. This can help the developer as well as other team members to better understand the potential risks of the code changes and better coordinate their programming tasks.

Some previous works [6], [1], [12], [13] have studied the role and influence of TODO comments in software development. There are also several approaches [14], [2], [11], [15], [16] proposed to analyze and maintain TODO comments. However, to the best of our knowledge, solutions that can perform automatic *TODO-missed commits* detection have not yet been developed. It is difficult to identify *TODO-missed commits* due to the following challenges:

Capturing software commit semantics: Identifying *TODO-missed commits* first needs to understand the semantics of the code changes. Sometimes it is difficult to determine whether a commit is a *TODO-missed commit* by just reading the changed code fragments. This requires developers to have enough knowledge of the code context as well as the existing implementations. An example is shown in the Example 2 in Figure 1, even though the code change is presented, one can not easily claim the code change is suboptimal due to his/her unfamiliarity with the code context. However, the associated commit message (i.e., *partial implementation of wiki article language preference*) provides some additional clues to fill up this gap. Therefore, if the information contained in both code changes and commit messages are taken into account, we can better capture the semantics of the software commits and make more accurate detections.

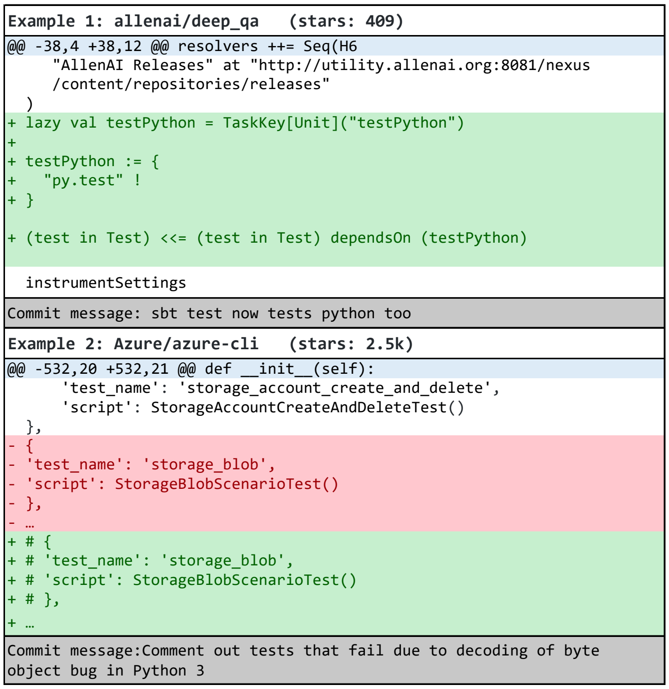
Dealing with imbalanced datasets: *TODO-missed commits* only account for a very small proportion of submitted commits. This will lead to the problem that the proportion of positive and negative samples is imbalanced when we build training data sets. According to our empirical study, the total ratio of the positive samples (i.e., *TODO-missed commits*) is only around 1.5% on average in all commits. For such imbalanced datasets, the performance of traditional classification algorithms will decrease dramatically. The traditional algorithms learn more from biased examples as opposed to the examples in the minority class. One might end up with a scenario where the model assumes that most test data belongs to the majority class. In order to better identify *TODO-missed commits*, it is not only necessary to construct a dataset suitable for classifier learning under imbalanced conditions, but also to make the algorithm more focused on hard, misclassified samples and prevent the vast number of easy negative samples from excessively affecting the learning process.

We present a novel approach named TDREMINDER (**TODO** comment **Reminder**) to automatically detect *TODO-missed commits*. Our approach has two phases: offline training and online inference. In the offline training phase, we collect commits from the top-10,000 Python and Java GitHub repositories, respectively. We select the commits that introduced TODO comments to the source code as the positive samples. The remaining commits are used as negative samples. Resampling is applied to make the training set more suitable for model training, while the test set can better reflect the performance in practice. Our approach is trained as a binary classification model with the dataset consisting of these two types of samples. With the aim of understanding the implementation of code changes more deeply, we take the knowledge of commit messages into account. To further capture the semantics of commits, we adapt the large-scale pre-trained model, namely CodeBERT [17], to encode the code changes and commit messages into contextualized vectors. In the training process of our model, Focal Loss [18] is selected as the loss function to aid better learning of hard samples and reduce the impact of simple negative samples. In the online inference phase, given a commit, our approach encodes code change and commit message respectively, and feeds them into the trained model. The model then outputs a score to determine whether the given commit is likely to be a *TODO-missed commit*.

In order to verify the performance of our approach, we conducted extensive experiments and a user study on Python and Java datasets. Since there are no previous studies investigating this problem, we constructed three baseline methods for comparative experiments. Our experimental results show that our TDREMINDER approach can significantly outperform all the baselines. The main contributions of our work include:

- We developed a novel tool, TDREMINDER, to automatically detect *TODO-missed commits* just in time. To the best of our knowledge, this is the first work that investigates the possibility of detecting *TODO-missed commits* in software repositories.
- We created a large-scale benchmark dataset of TODO-introducing commits, which is extracted from the top-10k Python and Java GitHub repositories respectively. As far as we know, it is the first and largest dataset for this task.
- We carried out extensive experiments using real-world popular repositories in GitHub, demonstrating the effectiveness and promising performance of TDREMINDER.
- We have released our source code of TDREMINDER [19]. The datasets used in our experiments have been made publicly available for community research.

The rest of this paper is organized as follows. Section 2 presents the motivating examples of our study. Section 3 describes the details of proposed approach. Section 4 describes



```
Example 1: allenai/deep_qa   (stars: 409)
@@ -38,4 +38,12 @@ resolvers ++= Seq(H6
     "AllenAI Releases" at "http://utility.allenai.org:8081/nexus
     /content/repositories/releases"
  )
+ lazy val testPython = TaskKey[Unit]("testPython")
+
+ testPython := {
+   "py.test" !
+ }

+ (test in Test) <<= (test in Test) dependsOn (testPython)

  instrumentSettings
Commit message: sbt test now tests python too
```

```
Example 2: Azure/azure-cli   (stars: 2.5k)
@@ -532,20 +532,21 @@ def __init__(self):
      'test_name': 'storage_account_create_and_delete',
      'script': StorageAccountCreateAndDeleteTest()
  },
- {
- 'test_name': 'storage_blob',
- 'script': StorageBlobScenarioTest()
- },
- …
+ # {
+ # 'test_name': 'storage_blob',
+ # 'script': StorageBlobScenarioTest()
+ # },
+ …
Commit message:Comment out tests that fail due to decoding of byte
object bug in Python 3
```

Fig. 2. Motivating examples

the data preparation for our approach. Section 5 shows the baseline methods, the evaluation metrics, the evaluation process and the experimental results. Section 6 shows the user study and its results. Section 7 is the discussion about TDReminder. Section 8 reviews the related work. Finally, Section 9 concludes the paper.

2 Motivation

Figure 2 shows two representative examples of *TODO missed commits* that we found in popular real-world repositories. Consider the first example in Figure 2. The submitter of this commit added a new code snippet about "python test task" to the repository. Through the associated commit message, we know that this developer added the Python test script to the "sbt test". But what the developer didn't realize is that if additional Python tests are added, the script will exit directly when the Python test fails. In other words, the original "sbt test" will not run completely because of the lack of exception handling mechanism due to the developer's mistake. We checked the modification history of this script and found that the developers were aware of this problem. In the next update, developers added a simple exception handling mechanism and introduced a TODO comment: "TODO(matt): it'd be nicer if this would still execute scala tests if python tests fail...". At the same time, they attached a commit message that says "testPython task now fails on failed tests". It seems that it was the failure of "testPython" task made them aware of this issue.

As shown in the above example, the loss of TODO comments may lead to varying degrees of defects. Some scholars have also found similar phenomena in previous studies on SATD [20], [5]. If TODO comments are not introduced in time, developers might only check the source code when the program throws an exception. Developers may often spend significant time locating the cause of a bug after a program crash, which can potentially impact software maintenance in terms of time and costs [21], [22], [23]. If a tool can help developers check whether their code changes are still suboptimal and/or incomplete before submitting the commit, this kind of situation will be avoided effectively.

The TODO comments can also help developers be aware of their code that needs to be further inspected to when making modifications or further implementations. For the Example 2 shown in Figure 2, the developers commented out some tests in the code change. If we only rely on the content of the code change, we are not sure why developers commented out these tests. The information in the commit message helps us fill this knowledge gap between source code and developer understanding. The decoding of byte object bug in Python3 makes it impossible for them to run these tests. Therefore, a TODO comment should have been introduced in the code to remind later developers. Once the formatter is fixed, developers should restore these tests immediately.

With the help of TDReminder, before developers submit their commits they will be reminded if a commit appears to be not complete and there is a need to introduce further TODO actions. Developers can add or modify comments or make further code modifications. This may help enhance code quality and potentially reduce the likelihood of encountering the problems exemplified above. With this proactive reminding about TODO comments, developers are unlikely to forget key pending tasks. After addressing the TODO comments, the reliability and maintainability of the system will also be increased.

3 Our Approach

In this section, we describe details of our proposed approach, TDReminder, to automatically identify the *TODO-missed commits*. In order to present our approach more clearly, we first define the task of *TODO-missed commits* detection. The overall framework is illustrated in Figure 3. The details of its offline training phase and online inference phase are described in Section 3.2.

3.1 Task Definition

Our goal is to **automatically identify a submitted commit that should have introduced TODO comment(s) but did not**. Once such *TODO-missed commits* are detected, our TDReminder tool then reminds developers to add corresponding TODO comments to the code. Given a commit, our approach needs to determine whether the commit is a *TODO-missed commit* or not. We formulate this task as a binary classification problem. For a given commit, we refer the code changes contained in the *diff* of the commit and the corresponding commit message as $\mathbf{x}$ and $\mathbf{c}$, respectively. The label of whether the commit is a *TODO-missed commit* or not is recorded as $\mathbf{y}$. We consider the label of a *TODO-missed commit* as positive, and the opposite as negative. Our approach will train a model θ using the training dataset. The probability $P_\theta\left(\mathbf{y}|\left\langle\mathbf{x},\mathbf{c}\right\rangle\right)$ is the conditional likelihood of predicting the label $\mathbf{y}$ with the given $\langle\mathbf{x},\mathbf{c}\rangle$. So the goal of our proposed model is to find:

$$\hat{y} = argmax_Y P_\theta\left(\mathbf{y}|\left\langle\mathbf{x},\mathbf{c}\right\rangle\right), \qquad (1)$$

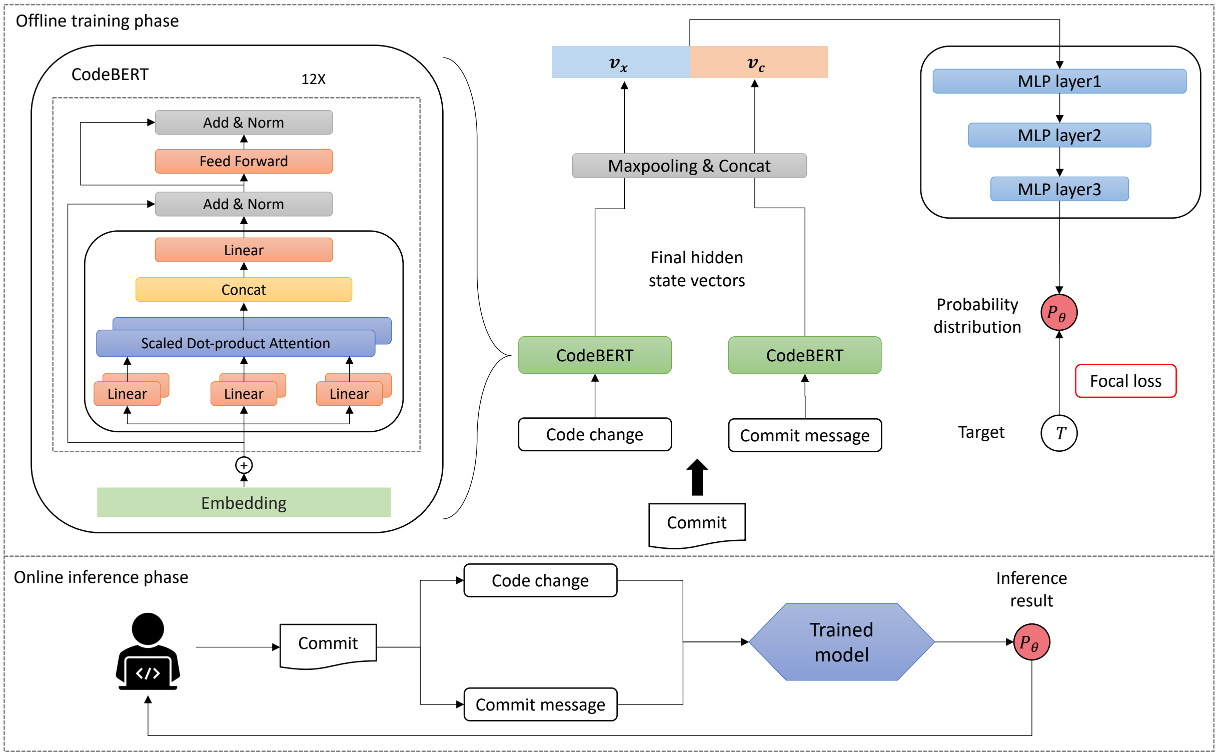


Fig. 3. The overall framework of our approach.

3.2 Implementation Details

There are several challenges in identifying such *TODO-missed commits*. As described in Section 1, the first challenge is to understand the code changes in the given commits. Given a commit, by only analyzing the code changes it will often be difficult for our approach to have a concept about the purpose of the code modifications. Thus, our approach introduces the information from the corresponding commit message to bridge the knowledge gap. TDReminder adopts two encoders to embed the code that has changed and its associated commit message into a vector representation. By learning from patterns in the training dataset, TDReminder has the potential to capture the connection between changed code and commit message.

However, another difficulty arises. The syntax of the code is fixed while the commit message is written in natural language. It is hard for traditional embedding techniques like word2vec [24] to embed both code and natural language as vectors. In order to capture the semantic correlation better, we employ CodeBERT [17] as the encoder component of our model. CodeBERT is a Transformer encoder which is pre-trained with natural-language descriptions and corresponding functions from open-source GitHub repositories. It is pre-trained for natural language and programming language. Previous studies [17], [25], [26], [27], [28], [29] have shown the effectiveness of CodeBERT for capturing semantics and context information of code and natural language. In this way, the code changes and commit message from the given commit can be transformed into the same vector space.

3.2.1 Encoders

As mentioned above, TDReminder introduces double encoders which are based on CodeBERT to enhance the understanding of commit data. Given a commit as input, we first extract the code changes $\mathbf{x} = \left\{x_1, \cdots, x_{|\mathbf{x}|}\right\}$ and the commit message $\mathbf{c} = \left\{c_1, \cdots, c_{|\mathbf{c}|}\right\}$ (shown in the training phase of Figure 3). These two sequences of tokens are then used as the input of *Code Change Encoder* and *Commit Message Encoder*, respectively. The structures of the two encoders are basically the same. Each encoder contains 12 layers of transformers. The hidden size of each layer is 768, and each layer has a self-attention sub-layer which is composed of 12 attention heads. After feeding the sequence of tokens into encoder, the encoder will calculate the contextualized representations $\mathbf{R}$, which includes final hidden states $\mathbf{H}$ of each token and the representation vector of special token [CLS]. In order to better represent the sequence, TDReminder applies the global max pooling [30] to compress the final hidden state vectors of tokens. The vector representation v obtained through the above calculation is used as the output of the encoder.

We take the input of code changes $\mathbf{x} = \left\{x_1, \cdots, x_{|\mathbf{x}|}\right\}$ as an example. The final hidden state vectors of encoder is $\mathbf{H_x} = \left[h_1, \cdots, h_{|\mathbf{x}|}\right]$. Thus, the contextual vector representation $v_{\mathbf{x}}$ of the code changes computed by the *Code Change Encoder* is as follows:

$$v_{\mathbf{x}} = maxpooling(\left[{h_i}^1, \cdots, {h_i}^k\right]), i \in \left[1, |\mathbf{x}|\right], \qquad (2)$$

where k denotes the hidden size of the final layer.

Correspondingly, the *Commit Message Encoder* embeds commit messages $\mathbf{c} = \left\{c_1, \cdots, c_{|\mathbf{c}|}\right\}$ into hidden state vectors $\mathbf{H_c}$. Likewise, the contextual vector representation $v_{\mathbf{c}}$ of the given commit message is calculated following Equation 2.

3.2.2 Multi Layer Perceptron

We use the information from commit messages to enhance the comprehension of the code changes made. However these code changes and their commit message are relatively independent when they are represented as contextual vectors. Therefore, we need to capture the relationships between them to boost the performance of our method. In order to address this, we employ a multi-layer perceptron (MLP) to integrate the information from code changes and commit message. The contextual vector representations $v_{\mathbf{x}}$ and $v_{\mathbf{c}}$ are first concatenated as a separate vector u. Then the multi-layer perceptron are applied to further extract information from the vector u. In this way, our model has the ability to capture the latent correlation between these features. In the end, the probability distribution of the final classification $\mathbf{y} \in \{0, 1\}$ is output by the last layer of MLP. More precisely, the probability distribution of the final classification $P_\theta\left(\mathbf{y} | \langle \mathbf{x}, \mathbf{c} \rangle\right)$ is computed as:

$$\begin{aligned}
&u = [v_{\mathbf{x}}; v_{\mathbf{c}}], \\
&\mathbf{r}_1 = a_1(W_1 u + b_1), \\
&..., \\
&\mathbf{r}_i = a_i(W_i \mathbf{r}_{i-1} + b_i), \\
&P_\theta\left(\mathbf{y} | \langle \mathbf{x}, \mathbf{c} \rangle\right) = \sigma(\mathbf{r}_i),
\end{aligned} \tag{3}$$

where W_i, b_i and a_i are the weight matrix, bias and activation function of the i-th layer, σ is the sigmoid function which can control the probability of the final output between 0 and 1.

3.2.3 Loss function

In our scenario, the number of positive samples is very much lower than that of negative samples. Unfortunately, there are a large amount of easy samples in these negative samples. The model will easily identify which category these easy samples belong to. At the same time, the trained model will then be more inclined to classify the test data into the category of these easy samples. Therefore, a model which focuses more on hard samples will be more helpful for our task.

To address this, we apply Focal Loss [18] to dynamically adjust the loss of each sample. Focal Loss is a loss function modified from the standard cross entropy loss function. The Focal Loss used in our approach is computed as follows:

$$\mathcal{L}(p_t) = -\alpha_t (1 - p_t)^\gamma log(p_t), \tag{4}$$

where t indicates the time steps, α is a weighting factor which controls the weight of different samples to the loss, γ is a focusing parameter which controls the degree of attention to the difficult samples. Compared with cross entropy loss, using a Focal Loss function can reduce the impact of easy samples on model training [18], so that our model will put more focus on hard, misclassified samples during training. In other words, we utilize Focal Loss to increase the learning rate of difficult cases, not just to mitigate the influence from class imbalance.

4 Data Preparation

We first introduce our data collection process from top-10k Python and Java software repositories (ordered by the number of stars). Even though our dataset is built from Python and Java datasets, our approach is language-agnostic and can be easily adapted for other languages. We then present our data construction process of dealing with the data imbalance problem.

4.1 Experimental Dataset Preparation

In order to build a large-scale dataset, we first clone the top 10,000 repositories from GitHub according to the number of stars. Repositories based on Java language or Python language were collected to mitigate the threats to external validity. Our study is not focused on one programming language, but rather on the development practices from the perspective of the entire repository. Many code changes in open-source repositories often involve cross-language collaboration [31], [32]. Therefore, we did not specifically collect projects that use one particular language, but instead selected projects that mainly use Java or Python as our data source. For each repository, we extract all the commits from the history. Every commit has a *diff* which represents the code change and a commit message describing the change. In addition to TODO comments, FIXME and XXX comments also indicate the existence of sub-optimized code implementations [3]. We need to use repositories with these types of code comment additions in their version history. We check all the commits of each repository, and if "TODO"/"FIXME"/"XXX" appears within any comment of their *diffs*, the corresponding repository is retained. Any repositories that have never introduced these types of annotations in historical submissions are excluded. After filtering, 5,467 Python repositories and 3,089 Java repositories remained. Many open-source projects do not allow developers to submit code changes which contain TODO/FIXME/XXX comments [3]. Some developers also do not want to commit their unfinished tasks in the form of these comments on open-source platforms [1]. These may be the causes why many of the repositories we collected do not contain TODO/FIXME/XXX comments.

After obtaining the required Github repositories, all of the commits from these repositories need cleaning and preprocessing. First, we extract all the *diffs* and commit messages from the commits of these repositories. Similar to the previous works [33], [16], [34] dealing with commits, we process *diffs* and commit messages respectively:

The commits with a *diff* larger than 1MB are all removed. Some *diffs* may introduce several TODO/FIXME/XXX comments. According to a previous study [16], this kind of commits are also removed because they are often used to batch update code comments. For the rest commits, *diffs* are extracted and converted into lowercase. Then we delete the *diff* header by using regular expressions. Commit IDs are replaced by "$\langle commit\_id \rangle$" to ensure semantic integrity. Finally, each *diff* is tokenized by blank space and punctuation marks.

We lowercase text in all commit messages. We retain the first sentences of the commit message since the first sentences are usually summaries of the entire commit message, as found by other researchers' prior work [35], [33]. For the same reason as *diff* cleaning, GitHub issue IDs and commit IDs are replaced by "$\langle issue\_id \rangle$" and "$\langle commit\_id \rangle$". Merge and rollback commits are removed by checking whether the

TABLE 1
Dataset summary statistics

Language	TODO (Positive)	FIXME	XXX	Negative	Total
Python	78,241	15,565	26,648	5,742,081	5,862,535
Java	72,467	9,239	7,210	4,747,915	4,836,831

commit messages began with "merge" or "rollback". We remove these commits for the reason that they obviously do not need to introduce comments to represent suboptimal implementations. Similarly, we tokenize the extracted commit messages with blank space and punctuation marks. In order to make our training and testing sets disjoint, we further remove the duplicated commits in the corpus.

During the data cleaning and filtering process mentioned above, approximately 31.35% and 34.32% of the commits in the remaining Python and Java repositories were deleted, respectively. Finally, we collected 5,862,535 and 4,836,831 cleaned data pairs from Python and Java repositories. Among them, each Python and Java repository contributes an average of 1,072 and 1,566 commits, respectively.

4.2 Dataset Construction

In this step, we marked all three types of code annotations simultaneously. Our goal is to identify the *TODO-missed commits*. The meanings and usage scenarios of FIXME/XXX comments are different from those of TODO comments [36], [37]. When submitting code changes, developers will choose different tags based on their different corresponding content of code change [38]. It is unreasonable to use these three categories together as positive samples. During the data processing process, we found that the number of FIXME and XXX was much lower than that of TODO comments. Based on the above two reasons, we do not add FIXME and XXX comments to the positive samples for now. Thus, the commits which introduce TODO comments are marked as **positive samples**. Then, the remaining commits from the same repository with positive samples are labeled as **negative samples**. We only focus on the commits that add new TODO/FIXME/XXX comments in the source code. This is reasonable because it is the code change in that commit that leads to the addition of a TODO/FIXME/XXX comment. Finally, we completed the labeling of the data and classified them into four categories. Table 1 shows the statistical results for each category.

Once the labeling task is completed, **we specifically delete TODO comments in all *diffs*, otherwise, our model can judge directly according to the addition and deletion of TODO comments.** When we detect a TODO comment within a *diff*, we further check if the subsequent lines are also comments. If so, these lines will be regarded as a part of the TODO comment. Finally, the TODO comment is deleted individually and the rest of the *diff* is tokenized again. The purpose of this process is to prevent the introduction of artificial traces indirectly caused by the deletion of TODO comments. For commit messages, we did not perform this cleaning operation. It is clear that our constructed dataset is extremely imbalanced, the number of the *TODO-missed commits* represents only a tiny percentage (i.e., around 1.5%) of all commits.

The primary data used in this study was drawn from the categories of "TODO (Positive)" and "Negative" in Table 1. To help ensure the dataset's quality, manual validation was performed to confirm the accuracy of labels applied to the samples. Given that both the datasets for Python and Java demonstrated an imbalance, the research implemented group sampling for each. We randomly selected 100 samples from both the positive and negative instances in each dataset, respectively. For each set, the first author meticulously checked whether each sample was assigned the appropriate label based on the corresponding code changes and commit messages. This involves identifying positive samples where the pending task has been resolved but the TODO comments remain unremoved, and identifying negative samples that contain either unfinished or suboptimal implementations.

With respect to the Python dataset, the verification results indicated that 6% of positive samples were incorrectly categorized, while a mere 1% of negative samples suffered from misclassification. In the case of the Java dataset, the misclassification rates for positive samples and negative samples were 5% and 2%, respectively. The results affirm that the overall accuracy of the labels in our provided dataset is reasonably high.

4.2.1 Intra-project Dataset

After getting the labeled data, we first construct an intra-project dataset using common general practices. We split all the data pairs into three sets: training set, validation set and test set. The imbalanced dataset may cause the performance degradation of the models if it is not handled properly. In order to ensure that the testing environment is similar to the actual working environment, we **do not** perform resampling when building the validation set and test set. This means the distribution of the validation and test set are the same as the real-world situations. We directly randomly select 50K samples as test set and validation set, respectively. For the training set, we then perform resampling according to a positive and negative sample ratio of 1:1. The purpose of resampling is to reduce the bias of our model learning caused by a large number of negative samples – this method is widely used to deal with imbalanced data [39], [40], [41], [42].

Finally, we follow the steps above to build our Python and Java datasets. We refer this dataset as *Intra-project Dataset*. We get 154k, 50k and 50k commit pairs in the training set, validation set and test set respectively for Python. For Java, there are 142k, 50k and 50k commit pairs in training set, validation set and test set respectively. Our model can learn and adjust the parameters from the common patterns contained in the training set. The validation set can help us avoid overfitting as much as possible. Finally, we can make detections on the test set to evaluate the performance of our trained model.

4.2.2 Inter-project Dataset

Due to random segmentation of all data pairs, similar commits from the same project may appear in both the training and testing sets. This may lead to deviations in the experimental results. LeClair et al. [43] also found a similar phenomenon in code summarization task. In order to investigate the inter-project performance of our proposed approach, we further construct a dataset which is split by project. We first count the amount of processed data contained in each remaining Java project and Python project. To ensure a data volume distribution similar to the *Intra-project Dataset*, we randomly select 1% of the projects and use all the processed data pairs in these projects as the validation set and test set. For the remaining projects, we extract all positive samples from them and resample the same number of negative samples to form the training set.

In the *Inter-project Dataset*, there are 154k, 52k and 35k commits pairs in training set, validation set and test set respectively for Python. For Java, we get 142k, 22k and 37k commit pairs in training set, validation set and test set respectively.

5 Empirical Evaluation

In this section, we present the setup and the results of our empirical evaluation. First, we describe the baselines, training settings and the evaluation metrics used in our experiments. We evaluate the performance of our approach TDReminder on the datasets described in Section 4. The research questions and the experimental results on *Intra-project Dataset* are described in Section 5.4. Then we investigate the inter-project performance of different methods in Section 5.5. Finally, we present some manual analysis of the results in Section 5.6.

5.1 Baselines

Since there is no previous work on detecting the *TODO-missed commits* that we could find, we have to construct baselines by ourselves. In order to compare the effectiveness of the proposed approach with other methods, we implement three baselines:

Random Guess: When there is no previous method for addressing the same research question, random guess is usually selected as a baseline [44], [45]. For each given commit, the Random Guess model will randomly determine whether it is a *TODO-missed commit* or not. Thus the evaluation metrics used in our experiments can be computed directly.

TODO Overlap Commit Message: It is a reasonable heuristic baseline which based on lexical overlap. Commit message is often the summary of the code change. If the submitter of the commit think there is a need to add TODO comments to the source code, he/she may mentions TODO related content in the commit message. Therefore, we manually read a large number of commit messages and collected some keywords with similar semantics. These TODO overlap keywords are as follows: “todo”, “unfinished”, “later”, “workaround” and “temporarily”. Then we create this baseline as follows: if any keyword is mentioned in the corresponding commit message of a commit, the commit is considered as a *TODO-missed commit*.

Random Forest based on Code Change: Random Forest (RF) [46] is an ensemble approach which combines many tree predictors. Each tree predictors in the forest depends on the values of a random vector sampled independently and with the same distribution for all these trees. Random Forest can often achieve good performance because it units many differently trained trees, it can mitigate overfitting and is not sensitive to outliers. Random Forest is also selected as one of the baselines in many works of software engineering [13], [47], [48], [49]. In our experiment, we first preprocess all the code changes by stop-word removal and tokenization. Then we embed the code change into vectors by word2vec technique [24]. At last, the mean of word embedding vectors are used as the input to the Random Forest algorithm. The Random Forest algorithm will output the probability distribution and its inference result for each commit.

5.2 Training Details

Our TDReminder approach was developed using the Pytorch framework in Python. The pre-trained model CodeBERT [17] is used as the encoder component for both code change and commit message. CodeBERT is easy to adopt in our approach. Due to the training design of CodeBERT model, it can handle source code and natural language well. While training the model, the dimension of final hidden state our encoders is 768. The output of encoders (*Code Change Encoder* and *Commit Message Encoder*) will be compressed into a 768 dimensional vector. The max length setting of the tokenizer is set to be 512, which is the maximum value of this optional parameter. The training batch size is set to be 16. We use AdamW [50] optimizer algorithm to optimize the parameters of our model with 2e-5 initial learning rate. The α and γ in Equation 4 is set to be 0.25 and 2.0 by default following the setting used by Lin et al. [18]. The maximum number of training epoch is set to be 10. We validate our model every epoch on the validation set. **The model which performs best on the validation set is used as the final trained model for evaluation.** We carry out all our experiments on a Ubuntu 20.04 server with one Nvidia A800 GPU and 80G memory, and 20 cores 3.7GHz CPU and 32GB memory.

5.3 Evaluation Metrics

Considering the highly imbalanced data (e.g., the positive samples is only around 1.5% on average) for our specific task, the traditional evaluation metrics (e.g., Precision and Recall) are not suitable for this usage scenario. We thus evaluate our approach with the following two evaluation metrics which are widely-adopted for unbalanced dataset evaluation [13], [51], [52].

AUC: AUC is a widely-used evaluation metric in many software engineering studies [13], [53], [51]. AUC represents the area under receiver operating characteristic (ROC) curve. The value of AUC ranges from 0 to 1, and higher AUC values indicate better performance, and a value of 0.7 is often considered as promising performance for different tasks. We choose the AUC metric for our study because of the following reasons: (a) AUC is robust towards imbalanced data distribution [53], [51]. The class distribution may affect some traditional evaluation metrics like precision, recall and F1-score and leads to unfair comparative experiments. Considering the dataset in our study is extremely imbalanced, we thus use AUC as one of the primary indicators due to its insensitive feature to class distributions. (b) AUC has the feature of statistical interpretable, for our case, AUC

TABLE 2
Comparison results with baseline models on different datasets

Language	Approaches	AUC	Cost-effectiveness
Python	Random Guess	50.0%	20.0%
	Commit message	52.1%	23.9%
	Random Forest	73.5%	50.0%
	TDReminder	**94.5%**	**91.6%**
Java	Random Guess	50.0%	20.0%
	Commit message	53.2%	26.3%
	Random Forest	74.3%	52.2%
	TDReminder	**95.2%**	**93.6%**

can be interpreted as the ability of a model to distinguish the positive cases (i.e., the *TODO-missed commits*) from the negative cases (i.e., the not *TODO-missed commits*). (c) AUC is threshold independent [54]. The setting of the threshold can determine whether the commit is labeled as positive or negative. However, in many cases like class imbalance case, the threshold can be changed according to the actual situation. Precision, recall and F1-score rely on the setting of the threshold, while AUC can independently evaluate the performance of the model.

Cost-effectiveness: Cost-effectiveness aims at maximizing the benefits by spending the same amount of cost, which has been widely used as an evaluation metric for defect prediction [55], [56], [52]. In our context, the cost is amount of commits to inspect, and the benefit is the number of *TODO-missed commits* that can be discovered. If we inspect all the inferred commits, the percentage of the *TODO-missed commits* is the recall. However, due to time constraint and/or limited resources, developers can only inspect a limited number of software commits. For this kind of situation, it is desirable to identify as many *TODO-missed commits* as possible while minimizing the number of commits to inspect, therefore, we introduce the cost-effectiveness evaluation metric for such cases. Particularly, the cost-effectiveness in our study denotes the recall of the *TODO-missed commits* when using 20% of the entire effort required to inspect all commits to inspect the top ranked commits.

5.4 Experimental Results

We want to answer the following three research questions:

- *RQ1: How well does our* TDREMINDER *perform against the baselines?*
- *RQ2: What are the impacts of the different components in our model?*
- *RQ3: How does focal loss affect the performance of our approach?*

Same as most of studies, we conduct extensive experiments on ***Intra-project Dataset*** to answer the above three research questions, results discussed below.

5.4.1 *RQ1: How does our* TDREMINDER *perform against baselines?*

Motivation. In order to evaluate the effectiveness of our proposed approach, we conducted a comparative experiment with our method and baselines. Due to the lack of previous work to address the detection of *TODO-missed commits*, we have constructed several basic methods as baselines. Compared to these baselines, we wanted to evaluate the overall performance of our TDREMINDER in terms of the AUC and Cost-effectiveness.

Approach. We compare TDREMINDER against the three baselines described in Section 5.1. For a better presentation of our experimental results, we refer to the baseline "TODO Overlap Commit Message" as **Commit message** and "Random Forest based on Code Change" as **Random Forest**. As described in Section 5.3, the selection of evaluation metrics needs to be suitable for the scenario of this work. To show the comprehensive classification ability of the models, we used AUC and cost-effectiveness in our experiments. These two metrics are more robust and appropriate to evaluate the performance in this task. We trained **Random Forest** and TDREMINDER on the training set and conducted inference on the test set.

Results. Table 2 presents the results of different methods applied to the Python dataset and Java dataset. From the table, we can see that **Random Guess** and **Commit message** have low AUC scores and cost-effectiveness scores. The performance of **Commit message** is slightly better than **Random Guess**. This phenomenon demonstrates that although **Commit message** can identify some *TODO-missed commits*, it still misses a lot of positive samples. From the results, we can see that the proportion of *TODO-missed commits* that can be identified by using the TODO keyword in commit messages is very small. On the other hand, when developers introduce a TODO comment into the source code, they will sometimes mention relevant content in the commit message when submitting the commit. The information contained in the commit message can help us detect *TODO-missed commits*, but we can not rely entirely on the commit message.

Random Forest based on Code Change, which combined with Random Forest and word2vec technique, performs the best among all the baselines on two datasets. The technique **Random Forest** baseline employed, Random Forest [46], is widely used in classification tasks in various fields and has achieved good results. In addition, unlike a simple method checking whether there is lexical overlap, **Random Forest** applies the word2vec technique [24] to capture the semantic information in the code changes. Word2vec is a popular technique for embedding text into a vector representation. Nevertheless, the word2vec technique still suffers from some serious issues. Word2vec technique can not capture dependency information over a long distance and can not deal with the problem of polysemy [57]. For example, when the same function name appears in two distant positions within a long *diff*, word2vec will not be able to capture the relationship. Some polysemous words that may appear in a commit, such as "pointer" (could refer to a programming "pointer" or a UI "cursor"), cannot be handled by word2vec. It's possible that these technique disadvantages could limit the effectiveness of **Random Forest**.

In our experiments, the improvement ratio of our approach is computed as $\frac{Ours-baseline}{baseline} * 100\%$. Compared to the **Random Forest** on the Python dataset, TDREMINDER outperforms **Random Forest** by 28.6% and 83.2% in terms of AUC and cost-effectiveness, respectively. As for the comparison experiments based on the Java dataset, the relative improvements of TDREMINDER are 28.1% and 79.3%

TABLE 3
Effectiveness of each components in TDREMINDER

Language	Approaches	AUC	Cost-effectiveness
Python	Drop-CC	75.3%	53.4%
	Drop-CM	93.1%	89.3%
	TDReminder	**94.5%**	**91.6%**
Java	Drop-CC	76.8%	55.4%
	Drop-CM	94.0%	90.2%
	TDReminder	**95.2%**	**93.6%**

w.r.t. AUC and cost-effectiveness, respectively. On average, TDREMINDER outperforms **Random Forest** by 28.4% and 81.3% in terms of AUC and cost-effectiveness respectively. Our approach achieves the best performance for all evaluation metrics in experiments. We attribute our good performance to the following three points: (1) We make full use of information from different resources, e.g. *diff* and commit message. The information from the commit message further enhances the understanding of code change. (2) **CodeBERT** is employed to further represent input into contextual vectors. For the reason that CodeBERT is pre-trained with natural language and source code, the *diff* and commit message can be transformed into the same vector space. Thus, our approach is able to better understand their semantics and connections, and then distinguish *TODO-missed commits*. (3) With the help of focal loss, TDREMINDER has better learning ability for hard samples.

Furthermore, we can observe that TDREMINDER is stable on both Python and Java datasets. This demonstrates the generalization of our approach. We believe TDREMINDER can support other programming languages just as well.

In summary, TDREMINDER obtains **higher AUC as well as cost-effectiveness scores than all the baselines on our two datasets, demonstrating the superior performance of our approach**.

5.4.2 RQ2: What are the impacts of different components in our model?

Motivation. To better capture the semantic features of the software commits, we adopt two encoders, i.e., *Code Change Encoder (CC Encoder)* and *Commit Message Encoder (CM Encoder)*, to encode the code change and the commit messages respectively. To verify the effectiveness of the two input components, we conduct a component-wise evaluation to evaluate their individual performance as well as their contributions one by one.

Approach. To answer this research question, we first build two incomplete versions of TDREMINDER:

1) **Drop-CM**: For a given commit, we only consider the code change of this commit and ignore the commit message, in other words, it only uses the *Code Change Encoder* and drops the *Commit Message Encoder*.
2) **Drop-CC**: For a given commit, we only consider the commit message as input and ignore the associated code change. In other words, we keep the *Commit Message Encoder* and drop the *Code Change Encoder*.

Then we train the above two variants and TDREMINDER using both *Commit Message Encoder* and *Code Change Encoder* according to the training details described in section 5.2. Finally, we compare the performance of these three approaches on the test set.

Results. The experimental results of the component-wise evaluation are shown in Table 3. It can be seen that:

- By comparing the performance of our approach with **Drop-CC** and **Drop-CM**, we can measure the performance improvements achieved by incorporating the commit messages and code change as inputs. For example, incorporating the information of commit messages, the AUC, Cost-effectiveness of our approach score increases 1.5% and 2.6% on Python dataset and 1.3%, 3.8% on Java dataset. By incorporating the code change, the AUC, Cost-effectiveness of our approach increases 25.5%, 71.5% on Python dataset and 24.0%, 69.0% Java dataset. It is clear that the code change do make a significant contribution to the overall performance of our approach. On this basis, the commit message further enhances the performance of the approach.
- To bridge the gap between status of the software commits and the code semantics, we introduce the *Code Change Encoder* and *Commit Message Encoder* components for capturing the features from code change and commit message respectively. **No matter which component we dropped, it hurts the overall performance of our model.** This also verifies the importance and necessity of incorporating the double encoder architecture for our model.

In summary, **both the *Code Change Encoder* and the *Commit Message Encoder* are effective and helpful to enhance to performance of our approach**.

TABLE 4
The impact of focal loss

Language	Approaches	AUC	Cost-effectiveness
Python	Drop-FL	93.4%	89.7%
	TDReminder	**94.5%**	**91.6%**
Java	Drop-FL	94.1%	89.9%
	TDReminder	**95.2%**	**93.6%**

5.4.3 RQ3: How does focal loss affect the performance of our approach?

Motivation. In our TDREMINDER, focal loss is applied to mitigate the influence from the large number of simple samples in our imbalanced datasets. Focal loss modifies the traditional cross entropy loss and makes the model pays more attention to the samples that are difficult to classify. In focal loss, the weight of loss brought from the hard, misclassified samples are enlarged, while the weight of the easy sample is decreased. It is widely used in various classification tasks, i.e., object detection, emotional analysis and text classification, and achieves good results. In this research question, we conduct an ablation study to analyze the performance gain achieved due to the focal loss.

Approach. In order to eliminate the impact of focal loss, we replace the focal loss with cross entropy loss function which is commonly used in classification algorithms. Then we retrain the TDREMINDER on the *Intra-project Dataset*

TABLE 5
Inter-project performance of different methods

Language	Approaches	AUC	Cost-effectiveness
Python	Random Guess	50.0%	20.0%
	Commit message	52.2%	24.3%
	Random Forest	68.8%	42.7%
	TDReminder-inter	**92.8%**	**86.9%**
Java	Random Guess	50.0%	20.0%
	Commit message	52.3%	24.5%
	Random Forest	61.2%	30.7%
	TDReminder-inter	**94.8%**	**89.8%**

again. To facilitate reading, **Drop-FL** is used to represent the variant that does not use focal loss in TDREMINDER.

Results. Table 4 shows the results. From the table, we can see that the performance of TDREMINDER is better than **Drop-FL** on both datasets. For Python dataset, TDREMINDER outperforms **Drop-FL** by 1.2% and 2.1% w.r.t. AUC, and cost-effectiveness, respectively. On Java dataset, TDREMINDER outperforms **Drop-FL** by 1.2% and 4.1% in terms of AUC, and cost-effectiveness, respectively. The average improvements are 1.2% and 3.1% in terms of AUC, and cost-effectiveness, respectively. In summary, **focal loss function is helpful and effective for the task of *TODO-missed commits* detection.** These experimental results verify the importance of focal loss in our proposed TDREMINDER method.

5.5 Inter Project Performance

Motivation. When merging data from all repositories and segmenting the dataset, there is a risk of similar or even identical samples appearing in the training, validation, and testing sets. For these data, the model will be able to make inferences easily. This will result in higher AUC and cost-effectiveness scores, resulting in deviation in model performance evaluation. When the model is used to make inferences on data from other projects, it may not perform as expected.

Approach. In order to study the generalizability of our approach, we further construct the *Inter-project Dataset* which is split by project (as described in Section 4.2.2). Then the proposed TDREMINDER and the baselines are retrained and tested on the *Inter-project Dataset*.

Results. Table 5 shows the inter-project performance of different methods. From the table, we can find that our approach TDREMINDER outperforms **Random Forest** by 34.9% and 103.5% in terms of AUC and Cost-effectiveness respectively on the Python dataset. As for the Java dataset, the relative improvements of TDREMINDER are 54.9% and 192.5% in terms of AUC and Cost-effectiveness, respectively. The performance of our approach TDREMINDER is still ahead of the baseline method **Random Forest**, and even more advanced on the *Inter-project Dataset* than on the *Intra-project Dataset*.

Compared with the results of intra-project performance in Section 5.4.1 (shown in Table 2), we can see that all methods except for heuristic-based have shown a decrease in performance. This phenomenon is similar to the findings from LeClair et al. [43]. The AUC and Cost-effectiveness scores of baseline **Random Forest** dropped 12.0% and 27.9% on average, respectively. For TDREMINDER, the AUC and Cost-effectiveness scores dropped 1.1% and 4.6% on average. Despite the performance penalty, our approach is relatively less degraded. There are two main reasons, the first is that the corpus we provide is broad enough for models to learn the features, and the second is the excellent representation and learning ability of the approach we propose.

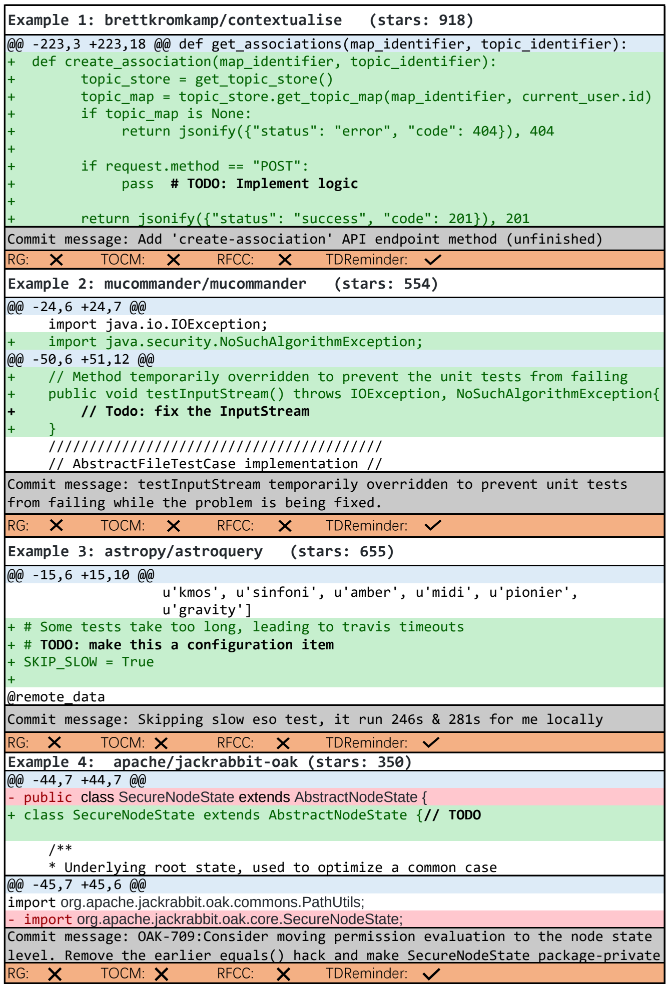
In conclusion, the experimental results on the *Inter-project Dataset* **show the good generalizability of our proposed TDREMINDER.**

5.6 Manual Analysis

In this section, we further discuss the reasons why our approach outperforms others by manually investigating some samples from the experimental results. TDREMINDER utilizes the information from *diff* and commit message for better performance of the model. Our experiments described above have shown the effectiveness of our approach. To further examine the performance of TDREMINDER, we manually inspected the inference results on our test set from different baselines.

Approach. For both Python and Java datasets, we randomly sampled 50 positive and 50 negative samples from their test sets. Subsequently, we obtained the inference results of various baselines and TDREMINDER for these samples. After carefully examining all the samples, four representative examples are selected to show in the qualitative results. Figure 4 shows the examples. **To better showcase the original intent of the code committers, we have reinserted the TODO comments that were removed from the *diffs* back to their original positions.** In the actual dataset, if a TODO comment does not occupy an entire line, as shown in the first and fourh examples of Figure 4, the TODO comment within that line will be individually deleted. If a TODO comment occupies a single line or multiple lines on their own, then these lines (including newline characters) will be removed together.

Results. In the first example, the developers submitted the commit without implementing specific logic under the "request.method == 'POST'" branch. All of the baselines, i.e., RG (Random Guess), TOCM (TODO Overlap Commit Message), and RFCC (Random Forest based on Code Change), are unable to accurately determine whether this commit is a *TODO-missed commit*. However, our method can successfully identify this commit as TODO-missed. We attribute its better performance to two reasons: (1) The pre-trained CodeBERT that TDREMINDER employed is well-trained for natural language and programming language. Furthermore, our model has been trained on the large-scale dataset and learned the common patterns from the semantic features. Thus, our model has a better semantic understanding of code changes and commit messages. (2) TDREMINDER absorbs the knowledge from the commit message. In this case, the commit message attached to the commit says, "Add 'create-association' API endpoint method (unfinished)". In the submission history of repositories, vocabulary representing unfinished semantics often appears simultaneously with pending tasks. Therefore, the



```
Example 1: brettkromkamp/contextualise   (stars: 918)
@@ -223,3 +223,18 @@ def get_associations(map_identifier, topic_identifier):
+  def create_association(map_identifier, topic_identifier):
+        topic_store = get_topic_store()
+        topic_map = topic_store.get_topic_map(map_identifier, current_user.id)
+        if topic_map is None:
+            return jsonify({"status": "error", "code": 404}), 404
+
+        if request.method == "POST":
+            pass  # TODO: Implement logic
+
+        return jsonify({"status": "success", "code": 201}), 201
Commit message: Add 'create-association' API endpoint method (unfinished)
RG:  ✗     TOCM:  ✗     RFCC:  ✗     TDReminder:  ✓
Example 2: mucommander/mucommander   (stars: 554)
@@ -24,6 +24,7 @@
     import java.io.IOException;
+    import java.security.NoSuchAlgorithmException;
@@ -50,6 +51,12 @@
+    // Method temporarily overridden to prevent the unit tests from failing
+    public void testInputStream() throws IOException, NoSuchAlgorithmException{
+        // Todo: fix the InputStream
+    }
     ////////////////////////////////////////
     // AbstractFileTestCase implementation //
Commit message: testInputStream temporarily overridden to prevent unit tests
from failing while the problem is being fixed.
RG:  ✗     TOCM:  ✗     RFCC:  ✗     TDReminder:  ✓
Example 3: astropy/astroquery   (stars: 655)
@@ -15,6 +15,10 @@
                   u'kmos', u'sinfoni', u'amber', u'midi', u'pionier',
                   u'gravity']
+ # Some tests take too long, leading to travis timeouts
+ # TODO: make this a configuration item
+ SKIP_SLOW = True
+
@remote_data
Commit message: Skipping slow eso test, it run 246s & 281s for me locally
RG:  ✗     TOCM: ✗      RFCC:  ✗     TDReminder:  ✓
Example 4:  apache/jackrabbit-oak (stars: 350)
@@ -44,7 +44,7 @@
- public class SecureNodeState extends AbstractNodeState {
+ class SecureNodeState extends AbstractNodeState {// TODO

     /**
     * Underlying root state, used to optimize a common case
@@ -45,7 +45,6 @@
import org.apache.jackrabbit.oak.commons.PathUtils;
- import org.apache.jackrabbit.oak.core.SecureNodeState;
Commit message: OAK-709:Consider moving permission evaluation to the node state
level. Remove the earlier equals() hack and make SecureNodeState package-private
RG:  ✗     TOCM: ✗      RFCC:  ✗     TDReminder:  ✓
```

Fig. 4. Manual Analysis Examples

model learned the association between these words and *TODO-missed commit* in the training phase. The keyword "unfinished" further helps our model identify this *TODO-missed commit*.

The second example shows a similar situation. Because we remove the TODO comments from the code changes when we build our datasets, TDReminder finds that there is nothing in the "testInputStream()" function in this case. The commit message and the comment in the code change both mentioned that this function is only temporarily overridden to prevent a test from failing. Based on this information, TDReminder can determine that the commit is still an incomplete implementation and there is a need to apply further TODO actions.

Our model has learned the code patterns of *TODO-missed commit* from the large training set, and the encoder component for commit messages further enhances the performance. Although certain keywords in commits can assist in identifying *TODO-missed commits*, some text from the commits may also mislead our model, and these commits sometimes do not need to introduce TODO comments at all. To validate our assumptions, we conducted probing like Karmakar et al. [58] to investigate whether keywords in commits can assist in the detection of *TODO-missed commits*. We randomly selected samples from the test set of the *Intra-project Dataset* that contained keywords such as 'unfinished', 'later', 'workaround', and 'temporarily' in their commit messages. For both Python and Java datasets, we retrieved 10 positive samples and 10 negative samples, respectively. Subsequently, we employed TDReminder to conduct inference on these samples. For the positive samples, which contained these keywords and were actually *TODO-missed commits*, TDReminder accurately identified all of them. Regarding the negative samples, TDReminder correctly inferred 6 out of the Java samples and 7 out of the Python samples. This shows that even if these keywords appear in negative samples and cause some misdirection, our model's understanding of code changes can mitigate the impact of this misguidance. These results, to some extent, reflect the correctness of our assumptions.

Upon inspecting the inference results, we observed that numerous test commits do not contain any prompt words in the commit message or source code. Nevertheless, our method accurately identifies these commits. For instance, in the third case illustrated in Figure 4, the developer introduced a "SKIP_SLOW" parameter to circumvent tests that are frequently timed out due to their prolonged execution. While this approach may not directly lead to system crashes, integrating the parameter into a configuration file would offer a more optimal solution, facilitating uniform settings across developers. In the fourth case in Figure 4, the developer make the `SecureNodeState` package private. According to our experience, it's essential to verify if associated calls in other packages have been removed. Otherwise, this will cause the relevant module to compilation failures.

In the above two examples, TDReminder successfully recognizes instances of suboptimal code implementation even without any prompts from relevant comments or commit messages. This capability arises from its training on expansive datasets. Such training enables the model to accurately identify similar situations based on the learned patterns, equipping it to detect code submissions that require further improvement. While the training data derives from scenarios of suboptimal implementations acknowledged by developers, the amassed development insights offer valuable assistance to the broader developer community. However, there are also some cases that our method can not handle well. When the code changes are too complicated, such as the modification of multiple submodules, it is often difficult for TDReminder to learn their semantic information. At this time, we may need to rely on the semantic features in commit messages. If the commit message is of low quality, TDReminder will not be able to make correct detections in such a situation.

6 User Study

In order to evaluate the performance of TDReminder more comprehensively, we choose metrics that are threshold independent and robust towards imbalanced data distribution in the above experiments. **In this section, we study the potential practical value of** TDReminder**.** While our proposed TDReminder can assist developers in identifying *TODO-missed commits*, it is more actionable for developers if we can also offer interpretable suggestions of "where" and "what" TODOs should be added. To this end, inspired by the remarkable performance of Large Language Models (LLMs) in natural language understanding and logic reasoning, we have integrated TDReminder with LLMs to construct a tool more tailored for software engineers. It is also an attempt to utilize TDReminder to build practical development plug-ins. Following this, we designed a user study to investigate the tool's efficacy in aiding developers to detect suboptimal code implementations. Our study design was approved by Zhejiang University's Ethics Advisory Board.

6.1 Tool Implementation

The overall architecture of the tool is shown in Figure 5.

TDReminder module: Given a commit submitted by a developer, we first extract its code change (represented by *diff*) and associated commit message. These are then fed into TDReminder. The TDReminder first determines whether this commit is a *TODO-missed commit*. If not, there is no need for further action. If it is, the commit is then input into the subsequent retrieval module.

Retrieval module: The retrieval module first preprocesses all the training corpus. Utilizing the trained *Code Change Encoder* which is described in Section 3.2.1, it represents every *diff* in the training set as vectors, facilitating similarity calculations. The collection of these vectors is denoted as T. Whenever a *TODO-missed commit* c is inputted, the retrieval module also uses the *Code Change Encoder* to represent its *diff* as a vector v_c. Subsequently, the module computes the cosine similarity between v_c and every vector in T. Finally, all commits from training set are ranked by similarity scores, and the most similar historical commit c_h is selected.

LLM module: After obtaining the historical commit record c_h most similar to the *TODO-missed commit* c, we utilize them to construct prompts for the LLM. The prompt we construct are as follows:

> You are an software engineer with 10 years of software development experience and is very familiar with the popular open-source software repositories on GitHub. Given a code changes (diff) and its associated commit message which contain a suboptimal implementation or a task that is yet to be completed. So we need to insert a TODO comment to highlight this. Here is an example:
> -diff: {diff of c_h}
> -commit message: {commit message of c_h}
> Please analyze and determine where and what TODO comment should be inserted for the following commit:
> -diff: {diff of c}
> -commit message: {commit message of c}

For convenience, we currently use gpt-3.5-turbo-0125 [1] as the base LLM for our implementation. The temperature value used for all API calls is set to be 0.2 [59]. Given the aforementioned prompt, it will analyze and suggest where and what TODO comments users should insert. In this manner, our user tool can seamlessly transition from identifying suboptimal code implementations to suggesting actionable TODO comments.

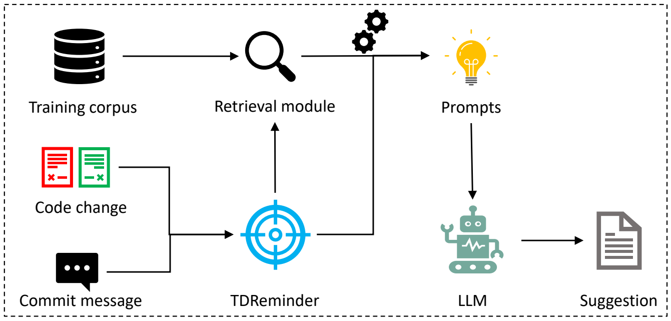


Fig. 5. The architecture of the user tool.

6.2 Study Design

To investigate the usefulness of our user tool in helping developers discover suboptimal code implementations, we conducted a user study. Through posting on the internal forum of Zhejiang University, we recruited a total of 40 volunteers to participate in our user research. These volunteers were required to have some development experience with Java or Python, and they were enthusiastic about using intelligent development plugins. Each participant was paid 40-80 Chinese Yuan when they completed the user study. This payment amount is comparable to the rates offered for various other experimental recruitments within the forum. Among them, 20 participants have industrial experience in Java programming ranging from two to six years (referred to as G1), and 20 participants have more than three years of Python industrial development experience (referred to as G2).

To mitigate biases from our imbalanced dataset, we randomly selected 50 samples, including 25 positive and 25 negative ones, from Java and Python test sets respectively as our evaluation data. Since we have already removed duplicate commits during dataset preparation (described in Section 4.1), the samples we drew from the test set would not coincide with the data in the retrieval corpus. To ensure that there were no mislabeled data in these samples, the first two authors manually checked the samples. Whenever we identified mislabeled data, we removed it and resampled another one from the remaining test set. G1 is required to evaluate the 50 commits of the Java programming language. Accordingly, the 50 Python code changes are evaluated by G2. To compare the effectiveness of different tools in assisting developers in identifying suboptimal code implementations, we further divided each group's 20 participants into four subgroups, namely subgroups A, B, C, and D.

In our user study, each participant first received a task guide. Participants were informed that they would receive variable payment depending on the number of correct answers that they make [60]. Then they were asked to read 50 commits one by one with suggestions from different tools and determine whether each commit needed to insert an additional TODO comment.

- For subgroup A, the provided suggestions come from the user tool which is referred as TDReminder.
- For subgroup B, we constructed a variant called TDReminder^{-r}. This variant eliminates the retrieval module from the TDReminder and directly requests the

[1] https://platform.openai.com/docs/models/gpt-3-5-turbo

LLM to generate suggestions based on the provided commits.

- As for subgroup C, the suggestions were based on baseline Random Forest because it performs best among various baseline methods. We directly provided the inference results of Random Forest to the evaluators.
- Subgroup D, serving as the control group, was not provided with any additional suggestions.

Then the participants only need to choose one answer from 'Yes' (additional TODOs need to be inserted), 'No' (no need to add additional TODOs), or 'Uncertain' (not sure if the commit is TODO-missing) for each sample. The time each participant spent to complete the questionnaire was recorded. Furthermore, the order of these samples in the questionnaire was shuffled, and participants were told that the provided suggestions may be not reliable. The participants were permitted to search the internet for any information.

Since the participants need to analyze 50 commits, there is a risk that participants may directly accept the provided suggestions without thinking on their own. Therefore, we added an additional attention check [61] in the questionnaire to mitigate its impact. We borrowed an idea of building the attention check from previous work [62], [63], [64]. We first sampled a new commit and added an extra statement at the end of its code change, "This is an attention check. Please select 'uncertain'." Then, for the provided suggestion in groups A, B, and C, we included some analysis text that is completely unrelated to the commit itself. For group D, suggestions were still not provided for reference. The attention check question appeared randomly in the questionnaire. In this way, participants who blindly answer questions or simply follow suggestions could be identified.

After completing these questions, participants of subgroups A, B, and C were required to answer two additional questions: 1) Please rate the usefulness of the suggestions provided on a scale of 1 to 5 (where 1 represents "not useful at all" and 5 represents "extremely useful"). 2) Do you think this tool could assist you in software development and maintenance? Please share your thoughts and briefly describe any specific benefits or potential issues that the tool might bring.

TABLE 6
Results of the User Study

Language	Suggestions	Accuracy (%)	Time (min)	Uncertain (%)	Usefulness
Python	No Suggestions	57.2 ± 5.4	33.5 ± 3.7	13.2 ± 3.9	-
	RandomForest	63.5 ± 7.7	27.7 ± 3.0	10.0 ± 4.9	2.0 ± 0.8
	TDReminder^{-r}	81.2 ± 7.4	30.1 ± 2.5	6.0 ± 3.2	3.8 ± 1.1
	TDReminder	86.4 ± 3.8	29.1 ± 3.1	3.6 ± 1.7	4.0 ± 0.7
Java	No Suggestions	55.6 ± 6.1	31.7 ± 3.0	16.0 ± 2.4	-
	RandomForest	66.0 ± 9.1	26.8 ± 3.5	14.0 ± 5.4	2.4 ± 0.5
	TDReminder^{-r}	82.4 ± 6.7	27.3 ± 2.2	6.4 ± 1.7	3.8 ± 1.1
	TDReminder	83.2 ± 5.9	27.6 ± 1.5	5.6 ± 3.8	3.8 ± 0.8

6.3 Results

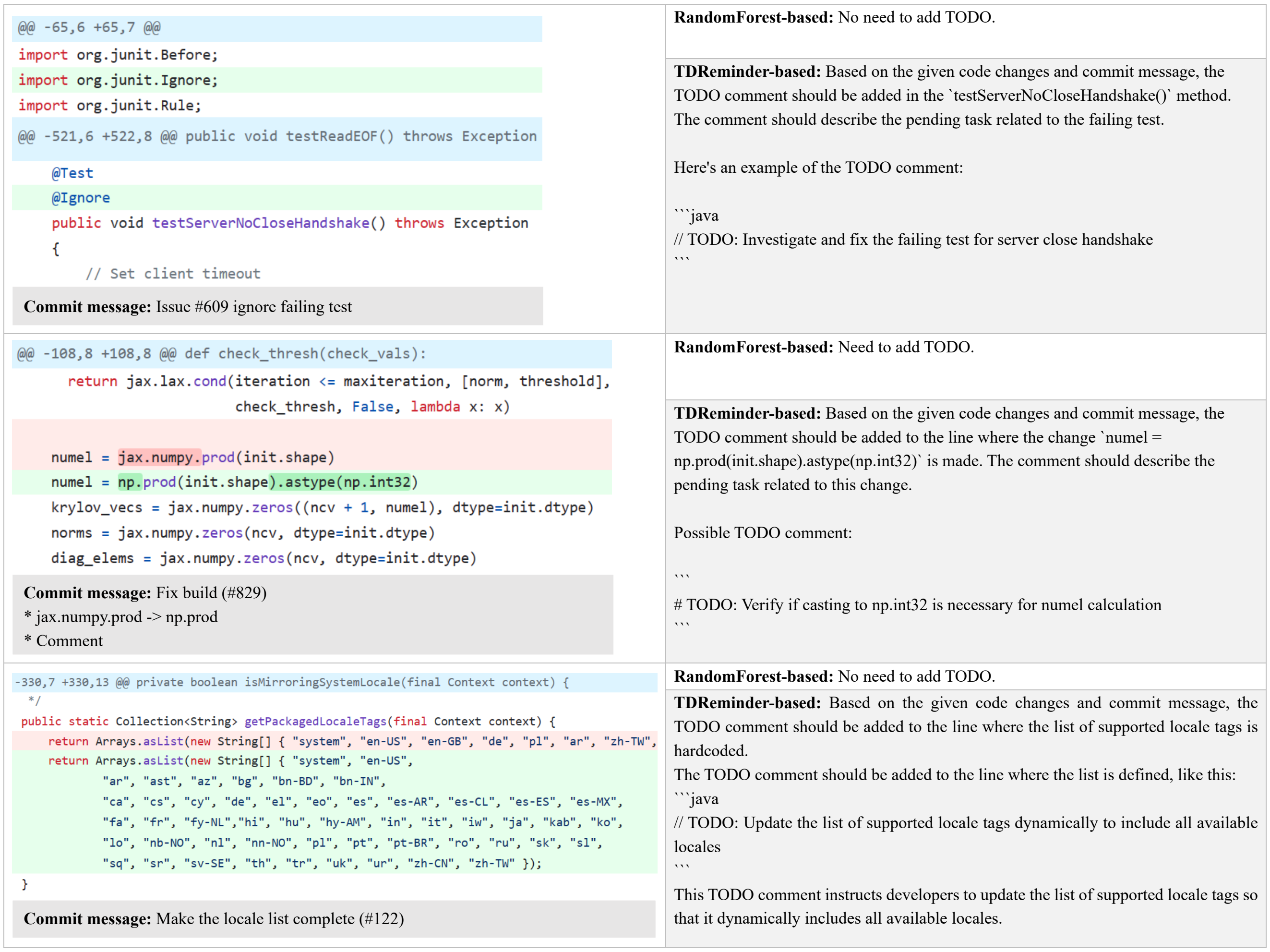
Consistent with previous work [62], [63], [64], we first identified participants who did not pass the attention test. In our user study, one person in the "RandomForest" subgroup of the Python language failed the attention check. This participant only received the lowest payment and their result was discarded. Table 6 presents the average accuracy, the completion time for the questionnaire, the proportion of "Uncertain" choices, and usefulness scores across different groups.

(1) From the table, it is evident that when developers were left to judge based solely on the content of the commits without any reference suggestions, the average accuracy for both Python and Java language samples was only 56.4%. With the suggestions from the baseline RandomForest, the average accuracy rate rose to 64.8%, though it still remained relatively low. In contrast, with the assistance of our user tool, participants' accuracy surged to 84.8% on average. In terms of accuracy, TDReminder outperforms the best baseline by 36.1% and 26.1% on Python and Java samples, respectively. Since our tool is a two-phase approach, we further examined the misclassification of samples by our tool and Random Forest, respectively. For TDReminder, it misclassified 7 out of 50 on both Python and Java samples. As for the tool based on Random Forest, the number of misclassifications on Python and Java samples is 15 and 16, respectively. Differences in accurately detecting suboptimal implementations may result in varied accuracy of participants' responses across different tools. We further conducted a Wilcoxon rank-sum test [65] to evaluate whether the differences between TDReminder and the above two baselines are statistically significant. The results showed that the improvements in our tool are all statistically significant at a 95% significance level. For the Python language, the p-values are 0.009 and 0.014 when comparing TDReminder with methods "No Suggestions" and "RandomForest", respectively. Similarly, for the Java language, the p-values are 0.009 and 0.012. This experimental result not only shows the excellent performance of our tool in detecting suboptimal code implementations but also demonstrates the usefulness of the suggestions provided by our tool.

(2) For the completion time, we can see that the control group that did not provide any suggestions took the longest time on average. Developers spent an average of 32.6 minutes on the questionnaire without the assistance of any tool. When utilizing recommendations from TDReminder, the time spent was reduced to an average of 28.4 minutes. The time for participants to answer each question is approximately 34.1 seconds. Compared to "No Suggestions", the completion time of TDReminder is shorter with statistical significance at the 95% confidence level on both Python and Java samples. Specifically, the p-values are 0.047 and 0.028, respectively. In addition, although the completion time of the "RandomForest" group is similar to that of the TDReminder, the low accuracy of this tool in detecting *TODO-missed commits* led to the participants in this group being misled.

(3) According to the uncertain rate in Table 6, our tool effectively decreased the rate of 'uncertain' choices by participants, indicating a reduction in their confusion. Compared to "No Suggestions", the assistance from our user tool has reduced the uncertainty ratio from 14.6% to 4.6% on average. Such findings highlight the tangible effectiveness gains that our proposed tool provides during software development and maintenance.

(4) For the rating of the usefulness of the tool by the participants, we can see that the average usefulness score



```
@@ -65,6 +65,7 @@
import org.junit.Before;
import org.junit.Ignore;
import org.junit.Rule;
@@ -521,6 +522,8 @@ public void testReadEOF() throws Exception

    @Test
    @Ignore
    public void testServerNoCloseHandshake() throws Exception
    {
        // Set client timeout
```

Commit message: Issue #609 ignore failing test

RandomForest-based: No need to add TODO.

TDReminder-based: Based on the given code changes and commit message, the TODO comment should be added in the `testServerNoCloseHandshake()` method. The comment should describe the pending task related to the failing test.

Here's an example of the TODO comment:

```java
// TODO: Investigate and fix the failing test for server close handshake
```

```
@@ -108,8 +108,8 @@ def check_thresh(check_vals):
      return jax.lax.cond(iteration <= maxiteration, [norm, threshold],
                          check_thresh, False, lambda x: x)

    numel = jax.numpy.prod(init.shape)
    numel = np.prod(init.shape).astype(np.int32)
    krylov_vecs = jax.numpy.zeros((ncv + 1, numel), dtype=init.dtype)
    norms = jax.numpy.zeros(ncv, dtype=init.dtype)
    diag_elems = jax.numpy.zeros(ncv, dtype=init.dtype)
```

Commit message: Fix build (#829)
* jax.numpy.prod -> np.prod
* Comment

RandomForest-based: Need to add TODO.

TDReminder-based: Based on the given code changes and commit message, the TODO comment should be added to the line where the change `numel = np.prod(init.shape).astype(np.int32)` is made. The comment should describe the pending task related to this change.

Possible TODO comment:

```
# TODO: Verify if casting to np.int32 is necessary for numel calculation
```

```
-330,7 +330,13 @@ private boolean isMirroringSystemLocale(final Context context) {
  */
 public static Collection<String> getPackagedLocaleTags(final Context context) {
     return Arrays.asList(new String[] { "system", "en-US", "en-GB", "de", "pl", "ar", "zh-TW",
     return Arrays.asList(new String[] { "system", "en-US",
             "ar", "ast", "az", "bg", "bn-BD", "bn-IN",
             "ca", "cs", "cy", "de", "el", "eo", "es", "es-AR", "es-CL", "es-ES", "es-MX",
             "fa", "fr", "fy-NL","hi", "hu", "hy-AM", "in", "it", "iw", "ja", "kab", "ko",
             "lo", "nb-NO", "nl", "nn-NO", "pl", "pt", "pt-BR", "ro", "ru", "sk", "sl",
             "sq", "sr", "sv-SE", "th", "tr", "uk", "ur", "zh-CN", "zh-TW" });
 }
```

Commit message: Make the locale list complete (#122)

RandomForest-based: No need to add TODO.

TDReminder-based: Based on the given code changes and commit message, the TODO comment should be added to the line where the list of supported locale tags is hardcoded.
The TODO comment should be added to the line where the list is defined, like this:

```java
// TODO: Update the list of supported locale tags dynamically to include all available locales
```

This TODO comment instructs developers to update the list of supported locale tags so that it dynamically includes all available locales.

Fig. 6. Three examples of the user study.

of the RandomForest-based tool is only 2.2. The average usefulness score of TDReminder reached 3.9. Compared to the RandomForest-based tool, our tool outperforms the baseline in terms of usefulness with statistical significance at the 95% confidence level. Specifically, the p-values are 0.020 for the Python samples and 0.028 for the Java samples. This is reasonable because our user tool can provide developers with actionable suggestions.

We examined feedback from participants regarding the RandomForest-based tool. Many comments highlighted a recurring issue: they were informed by a hint that the code was incomplete or suboptimal, which actually confuses them. Many participants expressed a desire for more specific information about the problem. Additionally, a few participants criticized that the tool was only useful in rare cases, such as when they previously remembered a sub-task but subsequently forgot about it. In such cases, the tool could remind users to recall the forgotten task before submission.

The feedback on our tool is more positive. 70% of participants agreed or strongly agreed that the tool could be beneficial in their daily development. Several participants found the majority of the recommendations to be sensible. They suggested that the tool could be beneficial during both the coding and code review phases to help identify overlooked issues. One participant mentioned that this tool can be combined with Linters, Formatters, and some code vulnerability scanning tools. A few participants expressed that although it may be difficult, it would be great if our tool could provide a fixed patch directly. However, several participants also pointed out that the suggestions in several examples were meaningless, and a few recommendations were irrelevant to what they considered to be suboptimal implementations. The comments from participant indicates the potential usability of our tool. Their thoughts also offer some insights into optimizing our tool and guiding future research directions.

(5) To investigate the effectiveness of the retrieval module introduced in our tool, we manually checked the suggestions generated by TDReminder^{-r} and TDReminder. Since we only generated suggestions for *TODO-missed commits*, we first selected the corresponding generated suggestions of these samples separately. Subsequently, our first author compared the original TODO comments and the generated suggestions for each sample. We did not strictly use “exactly match” as the evaluation metric, but instead marked this sample as long as two comments point to the same problem or describe similar pending tasks. For both Python and Java positive samples, 20 *TODO-missed commits* were correctly classified. For Python language, the proportion of marked suggestions generated by TDReminder and TDReminder^{-r} is 60% and 40%, respectively. For Java language, the cor-

responding proportions are 65% and 55%, respectively. As can be seen, with the addition of the retrieval module, our user tool can generate more suggestions that are close to the original TODO comments in our samples. Many previous studies have also reported that similar examples retrieved can help LLMs generate higher-quality answers [66], [67], [68]. However, in our study, the differences experienced by participants were not significant. In the future, we will combine our method with LLMs to construct a better framework to generate higher-quality suggestions or patches for suboptimal implementations.

6.4 User Tool Cases

To better illustrate the utility of our tool, three examples from positive samples are presented in Figure 6.

In the first example, the developer ignores the failing test temporarily. If it is not addressed, the test case may be missed in future testing. As can be seen from the figure, the Random Forest based tool failed to identify this *TODO-missed commit*. However, our tool not only accurately detected it but also provided targeted recommendations. The original TODO comment for this commit was ”work out why this test is failing”, while the suggestion we provided is more specific.

For the second case, the committer cast the data type of “numel” to “np.int32”. By searching for raw data, the original TODO comment we removed highlighted a potential issue: “check if this runs on TPU (dtype issue)”. From the recommendation given by the TDReminder-based tool, it can be seen that our tool also identified the suboptimal implementation and pointed out the potential issue related to data type at this location.

In the third instance, the developer hard-coded a fixed list of values. Directly embedding a list of constants (or any other values) in the code is often considered poor programming practice. It can hinder the reusability, flexibility, and maintainability of the code. Our tool recommends a dynamic updating approach as a more elegant solution. We also compared the original TODO comment from this commit, and its intended meaning aligns with this suggestion.

Our proposed approach, TDREMINDER, could effectively detect *TODO-missed commits* during code submissions. By utilizing the accompanying user tool, developers can receive a timely analysis of potential suboptimal implementations and be prompted to add TODO comments. Such guidance will **be helpful in improving the code and enhancing the comprehensiveness of the documentation in the software repositories.**

7 DISCUSSION

In this section, we present the threats to validity and some implications from this work.

7.1 Threats to Validity

Threats to internal validity are related to potential errors in the code implementation and experimental settings. We have double checked the code of our approach and the baselines. The baseline **TODO Overlap Commit Message** is easy to implement. As for the baseline **Random Forest based on Code Change**, we directly use the open-source software library scikit-learn [69]. The parameters used in our experiments are fine-tuned through many attempts. Basically, the parameter settings are considered and tested. For the reason that pre-trained CodeBERT has fixed the dimension of output, thus we use the same hidden size in our approach.

The second key threat to internal validity relates to the potential inaccuracies in our labeling process. In our study, the labeling of data is based on comment tags in commits, which means there may be unmarked suboptimal code implementations in some negative samples. As a result, the overall accuracy and reliability of the model in practical applications could be compromised. To mitigate this threat, we conducted manual validation during dataset construction to examine the accuracy of the datasets. The results of this manual validation indicate that the accuracy of the labels in our datasets is reasonably high. Therefore, the impact of this threat is very limited.

Threats to external validity relate to the generalizability of our experimental results. Because Python and Java are the most popular programming languages, we only use the commits from top-10K Python and Java repositories in GitHub to construct our datasets. But we believe it is not difficult to support other languages based on the approach we propose. Our model is not specifically designed for Python and Java languages. The experimental results in Section 5.4 on different language datasets also show that TDREMINDER is language-independent. In order to mitigate these threats, we will collect more data consisting of commits written in various programming languages and try to investigate our approach with such extended data.

The second key threat to external validity is that we removed the *diffs* which contain multiple TODO comments, following the approach of Gao et al. [16]. This was because such commits are often comment updates and will introduce noise into our datasets. Therefore, if TDREMINDER encounters commits with multiple TODO comments in the real-world scenario, our method may not work. In the future, we will try to overcome this shortcoming and make TDREMINDER a more universal tool.

Another key threat to external validity relates to the diverse practices of annotation in software development. Developers and development teams vary widely in their use of TODO comments. Some open-source software projects, for instance, prohibit contributors from submitting code with TODO comments [70]. Some developers believe that issue trackers are the appropriate place for TODOs, rather than embedding them in the source code [71]. Contrarily, research by Storey et al. [1] indicates that other developers find it is costly to add TODO comments to issue trackers. They tend to directly insert descriptions of subtasks into source code in the form of Technical Debt. During our data collection, we found that only 54.7% of the Python repositories and 30.9% of the Java repositories contained TODO/FIXME/XXX comments. It is clear that these limitations on writing Technical Debt significantly affect the comprehensiveness of our dataset. This resulted in over half of the total 20,000 projects we collected being excluded. Nevertheless, TDREMINDER still outperforms the baselines on both intra-project and inter-project datasets. The lack of some data does not significantly damage the performance of TDREMINDER. In terms of the actual impact of tools on developers, our user study shows that our tool can help them more accurately and quickly detect suboptimal implementations, thereby improving development efficiency and code quality. To further mitigate this in the future, we plan to expand our training dataset to include small-size repositories from GitHub and other open-source platforms.

Additionally, the filtering out of FIXME/XXX comments may also introduce a key threat to the external validity. The primary reasons for the filtering include that the different types of comments are associated with different content of code changes, and the quantity of FIXME/XXX comments is much lower compared to TODO comments. We have collected a large range of open-source repositories to increase the scale and variety of our data, and the experimental results have demonstrated the good performance of our model. However, this filtering operation may still limit the generalizability of our model. In the future, we plan to further investigate the real roles and practical characteristics of comments with different tags in software repositories. This will also facilitate our work on generalizing the model to handle different suboptimal implementation.

Threats to construct validity relate to suitability of our evaluation metric selection. We use AUC and cost-effectiveness to evaluate the effectiveness of our approach and baselines in our experiments. The dataset is highly imbalanced in this task. Thus the selection of evaluation metrics needs to be class distribution and threshold independent. The traditional metrics like recall and precision is not suitable for this situation. We pay more attention to the comprehensive performance of the model. AUC and cost-effectiveness are widely used in many past software engineering studies [55], [56], [52], [72], [13].

Another threat to construct validity relates to the user study in Section 6. We cannot guarantee that all participants' answers are given under the same evaluation criteria. Participants' programming experiences will also lead to their different understanding of the code. To mitigate this threat, we trained the participants and provided detailed examples and tutorials for them. We evaluated each case with 20 experienced participants who were interested in intelligent development tools, and they were provided with different reference information. Additionally, we applied an attention test to filter out participants who might blindly answer questions or follow suggestions without thoughtful consideration. We also introduced variable payment to encourage participants to search for useful information and carefully consider each question. Therefore, we believe that this threat has been mitigated to a considerable extent. In the future, we plan to develop a more intelligent tool and integrate it into popular IDEs (such as Visual Studio Code) to attract a larger number of experienced developers to evaluate our tool.

7.2 Implications for Practice

Understanding the context of code change: The proposed tool TDREMINDER is designed to detect missing TODO comments on change-level. TDREMINDER reduces a lot of effort to figure out which source file TODO comments need to be added. When these valuable contexts are provided to developers, they will be able to quickly locate the code snippets that need to be changed, and introduce TODO comments or further optimize the code implementation. As detailed in Section 7.1, the annotation practices among developers vary considerably. Based on the context, such as personal development, peer collaboration, or community involvement, developers may choose different strategies on using TODO comments. Some might even opt not to include TODO annotations in their code changes. While developers may not add TODO comments in some situations, our tool can still alert them to the presence of suboptimal code implementations and encourage them to improve code quality. Developers can also improve their development habits in this process, so as to make the software system develop healthily.

Ensuring comprehensive and timely inspections: Developers utilize TODO comments as reminders for themselves or teammates regarding code modifications. Using TDREMINDER, some missing TODO comments will be able to be added to the appropriate position in time. This ensures that some suboptimal code implementations are identified and valued by developers as much as possible. However, it is frequently observed that while TODO comments are added, they aren't addressed in time and sometimes are even overlooked. Many TODO comments are never formally migrated to change requests and remain obscured within the codebase for prolonged periods [1]. For such cases, the added TODO failed in their primary role to remind themselves or anyone else to actually update the code. Therefore, for software development practice, it is better to provide a mechanism to ensure that developers can revisit the unfinished TODO tasks in time. For instance, raising an error or exception when the associated class or function is loaded can serve as an effective reminder. In this way, TODO comments are not only added when necessary but also revisited and resolved at crucial stages in the development cycle. This allows the full value of TODO comments to be realized.

7.3 Implications for Research

For the task of *TODO-missed commits* detection, TDREMINDER has achieved promising performance on our large real-world Java and Python datasets. There are still some work we can do to further improve our approach. For example, one of the key problems is that we lack the context of the various libraries used in the project. If TDREMINDER could obtain some knowledge about the libraries, our approach may has the ability to know the relative risks of some API calls and remind developers to modify them in the future. Besides, other features about the code changes are also factors that can be considered in future methods. We could combine the change features (i.e., distribution of modified code across each file, number of modified subsystems and files, number of unique changes to the modified files before) in TDREMINDER and study the effect of various features on this task.

TODO comments are widely used in software development, and the life cycle of TODO comments involve many challenges, e.g., *where-TODO*, *what-TODO*, and *how-TODO*. For example, as mentioned above, revisiting and fixing the unfinished TODO comments (relevant to *how-TODO*) is meaningful and useful for developers. Our work of identifying *TODO-missed commits* first investigates the possibility of solving *where-TODO* tasks, which can benefit other researchers to further explore *what-TODO* and *how-TODO* tasks in the furture. This will make an important contribution to the management of TODO comments and suboptimal code implementations.

8 Related Work

This section discusses the related work with respect to TODO comments, SATD in software engineering as well as the mining software commits.

8.1 TODO comments in SE

TODO comments are extensively used by software developers during software development, and previous studies have investigated the practical usage of TODO comments in software engineering tasks.

Storey et al. [1] explored how the task annotations (i.e., TODO comments) can be used to support the work practices within the software development. They conducted a survey of professional developers, and they found that TODO comments make up a majority of the task annotations and developers often write TODO comments to mark the part of the source code which need their attention. Nie et al. [2] surveyed the importance of improving and the maintenance of TODO comments, and developed a framework to write trigger-action TODO comments in executable format. Sridhara et al. [14] presented a technique to check the status of the TODO comments, given a method associated with a TODO comment, their approach automatically checks if the TODO comment is up to date via using the information retrieval, linguistics and semantics methods. Most recently, Gao et al. [16] proposed a neural network based model, named TDCleaner, to detect the obsolete TODO comments from mining the commit histories of the software repositories. Wang et al. [73] investigated the quality characteristics of TODO comments and the lifecycle of these comments under varying quality conditions. Additionally, they developed a tool designed to differentiate TODO comments based on their quality.

Different from the aforementioned studies, our work focus on a novel task of detecting *TODO-missed commit* just-in-time during software development. To the best of our knowledge, this is the first research to exploit the possibility of automating the determination of *TODO-missed commit*.

8.2 Self-admitted Technical Debt in SE

Technical debt (TD) refers to developers taking suboptimal solutions to achieve short-term goals that may affect long-term software quality [4], [5]. The impact of self-admitted technical debt (SATD) on software development has garnered significant attention. For example, Kamei et al. [74] found that 42% to 44% of TD incurs positive interest by analysing Apache JMeter project. Russo et al. [20] shows that SATD contains many different weaknesses that may affect the security of the project. Wehaibi et al. [5] noted that while the presence of SATD often leads to complex changes, code changes with SATD surprisingly introduce fewer future defects compared to those without. This insight underscores the long-term value of detecting suboptimal implementations in minimizing code defects.

Prior works have investigated different ways to identify TD through source code [75], [76], [77] as well as source code comments [3], [78], [79], [74], [80]. Potdar et al. [3] proposed the self-admitted technical debt (SATD) concept (e.g., TODO, FIXME, and HACK) for the first time, which refers to the TD introduced by a developer intentionally and documented by source code comments. Huang et al. [79] proposed text-mining based methods to predict whether a comment contains SATD or not. Ren et al. [80] proposed a CNN-based approach for classifying code comments as SATD or non-SATD. Instead of detecting SATD at the file-level, Yan et al. [13] first presented the idea of "change-level SATD determination", which determines whether a change introduces SATD or not. More recently, Rungroj et al. [81] presented an approach to detect the "on hold" SATD in software repositories.

Some researchers have also been dedicated to the removal of SATD [7], [8], [9], [10]. da Silva Maldonado et al. [10] found that a significant portion of SATDs were removed by the creators themselves. After studying the relationship between SATD removal and code changes, Zampetti et al. [7], [8] proposed a deep learning-based classifier SARDELE to recommend six different SATD removal strategies. Liu et al. [9] analyzed the patterns in which different types of SATD were introduced and removed by investigating 7 deep learning projects.

In addition, there are also some works that focus on different aspects of the SATD practice [70], [71], [82], [83]. For example, Fucci et al. [82] analyzed 5 Java open-source projects and found that SATD was not necessarily introduced by code changes, but could also be introduced through code reviews by individuals with higher levels of project ownership. Xavier et al. [83] studied the proportion of SATDs solved in issue tracker systems and argued the need for better SATD management tools. Zampetti et al. [71] found that the admission level of SATD is similar in open-source and industrial communities, and is constrained by organizational guidelines. Cassee et al. [70] studied the polarity of SATD content and the situations in which developers would highlight the importance of SATD.

In this research, we primarily focus on the TODO-related commit for two main reasons. Firstly, the TODO comment is one of the most common SATD, which makes up a majority of the SATD. Secondly, aligned with our goal of assisting developers in the identification of suboptimal commits just in time, TODO-related commits often describe the functionality that needs to be paid attention to, making them particularly relevant for our investigation.

8.3 Mining software commits

Software development process generates a huge amounts of commits, and these commits are valuable resources during the software evolution. Prior works have investigated different software engineering tasks from mining these commits.

For example, Rosen et al. [84] presented a tool, named Commit Guru, to identify the risky software commits. Jiang et al. [33] adapted the neural machine translation techniques to the task of commit message generation. They presented a tool to automatically translate diff into commit messages. Following this work, Wang et al. [34] extended this research by training a context-aware encoder-decoder model for commit message generation. Liu et al. [85] proposed an approach to automatically generate descriptions for a pull request by considering the commit messages and code comments. Yan et al. [86] developed a two-phase framework to perform the defect identification and localization by exploring a total of 177K code changes during software development.

Different from the existing research, our research targets a novel commits related software engineering task, i.e., identifying the *TODO-missed commit* by mining the accumulated software commits histories. We have released the first dataset for this task to facilitate other researchers to extend our work and verify their own ideas.

9 Conclusion and Future Work

In this paper, we aim to automatically identify the *TODO-missed commits* whenever the developers submit code change. The novel approach named TDReminder (**TODO** comment **Reminder**) is proposed to detect *TODO-missed commits* and remind developers of the suboptimal changes. Our approach leverages both information contained in code changes and also the knowledge in commit messages to improve the effectiveness of TDReminder. Furthermore, the focal loss is employed to strengthen the model learning ability of hard, misclassified samples. In order to evaluate our approach, we build Python and Java datasets by collecting data from the top-10k Python and Java repositories in GitHub, respectively. Extensive experiments on the large-scale real-world datasets have demonstrated the effectiveness and performance of our approach. We believe that TDReminder will help developers better manage the potential risk of commits and thus improve the quality of software. We have made the code for our approach and datasets publicly available for community research. We plan to evaluate our TDReminder on commits written in other programming languages. We also plan to explore better characterization technique for commits and incorporate more knowledge about commits to further improve the performance. Finally, we will incorporate it into an IDE plugin and further gather developers' feedback in the wild.

Acknowledgements

This research is partially supported by the National Natural Science Foundation of China (No. 62302430 and No. 62202341), Zhejiang Provincial Natural Science Foundation of China (No. LQ24F020017), Zhejiang Province "JianBingLingYan+X" Research, the Starry Night Science Fund of Zhejiang University Shanghai Institute for Advanced Study (Grant No. SN-ZJU-SIAS-001), and Shanghai Sailing Program (23YF1446900). Grundy is supported by ARC Laureate Fellowship FL190100035. This work is also supported by the National Research Foundation, under its Investigatorship Grant (NRF-NRFI08-2022-0002). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

This work is supported by Zhejiang Provincial Engineering Research Center for Real-Time SmartTech in Urban Security Governance. The numerical calculations in this paper have been done on the supercomputing system in the Supercomputing Center of Hangzhou City University.

References

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

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

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

[4] Zengyang Li, Paris Avgeriou, and Peng Liang. A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 101:193–220, 2015.

[5] Sultan Wehaibi, Emad Shihab, and Latifa Guerrouj. Examining the impact of self-admitted technical debt on software quality. In *2016 IEEE 23Rd international conference on software analysis, evolution, and reengineering (SANER)*, volume 1, pages 179–188. IEEE, 2016.

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

[7] Fiorella Zampetti, Alexander Serebrenik, and Massimiliano Di Penta. Was self-admitted technical debt removal a real removal? an in-depth perspective. In *Proceedings of the 15th international conference on mining software repositories*, pages 526–536, 2018.

[8] Fiorella Zampetti, Alexander Serebrenik, and Massimiliano Di Penta. Automatically learning patterns for self-admitted technical debt removal. In *2020 IEEE 27th International conference on software analysis, evolution and reengineering (SANER)*, pages 355–366. IEEE, 2020.

[9] Jiakun Liu, Qiao Huang, Xin Xia, Emad Shihab, David Lo, and Shanping Li. An exploratory study on the introduction and removal of different types of technical debt in deep learning frameworks. *Empirical Software Engineering*, 26:1–36, 2021.

[10] Everton da S Maldonado, Rabe Abdalkareem, Emad Shihab, and Alexander Serebrenik. An empirical study on the removal of self-admitted technical debt. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 238–248. IEEE, 2017.

[11] Pengyu Nie, Junyi Jessy Li, Sarfraz Khurshid, Raymond Mooney, and Milos Gligoric. Natural language processing and program analysis for supporting todo comments as software evolves. In *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[12] Dorsaf Haouari, Houari Sahraoui, and Philippe Langlais. How good is your comment? a study of comments in java programs. In *2011 International Symposium on Empirical Software Engineering and Measurement*, pages 137–146. IEEE, 2011.

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

[14] Giriprasad Sridhara. Automatically detecting the up-to-date status of todo comments in java programs. In *Proceedings of the 9th India Software Engineering Conference*, pages 16–25, 2016.

[15] Innobuilt Software LLC. All your todo comments in one place. https://imdone.io, 2019.

[16] Zhipeng Gao, Xin Xia, David Lo, John Grundy, and Thomas Zimmermann. Automating the removal of obsolete todo comments. *arXiv preprint arXiv:2108.05846*, 2021.

[17] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[18] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*, pages 2980–2988, 2017.

[19] Tdreminder. https://doi.org/10.5281/zenodo.5402956, 2023.

[20] Barbara Russo, Matteo Camilli, and Moritz Mock. Weaksatd: Detecting weak self-admitted technical debt. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 448–453, 2022.

[21] Basma S Alqadi and Jonathan I Maletic. An empirical study of debugging patterns among novices programmers. In *Proceedings of the 2017 ACM SIGCSE technical symposium on computer science education*, pages 15–20, 2017.

[22] Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. How long will it take to fix this bug? In *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)*, pages 1–1. IEEE, 2007.

[23] Francesco Lomio, Emanuele Iannone, Andrea De Lucia, Fabio Palomba, and Valentina Lenarduzzi. Just-in-time software vulnerability detection: Are we there yet? *Journal of Systems and Software*, 188:111283, 2022.

[24] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[25] Jimmy Lin, Rodrigo Nogueira, and Andrew Yates. Pretrained transformers for text ranking: Bert and beyond. *arXiv preprint arXiv:2010.06467*, 2020.

[26] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

[27] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

[28] Jinfeng Lin, Yalin Liu, Qingkai Zeng, Meng Jiang, and Jane Cleland-Huang. Traceability transformed: Generating more accurate links with pre-trained bert models. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 324–335. IEEE, 2021.

[29] Maryam Vahdat Pour, Zhuo Li, Lei Ma, and Hadi Hemmati. A search-based testing framework for deep neural networks of source code embedding. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 36–46. IEEE, 2021.

[30] Yoon Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

[31] Philip Mayer and Alexander Bauer. An empirical analysis of the utilization of multiple programming languages in open source projects. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, pages 1–10, 2015.

[32] Philip Mayer, Michael Kirsch, and Minh Anh Le. On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers. *Journal of Software Engineering Research and Development*, 5:1–33, 2017.

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

[34] Haoye Wang, Xin Xia, David Lo, Qiang He, Xinyu Wang, and John Grundy. Context-aware retrieval-based deep commit message generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(4):1–30, 2021.

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

[36] Python developer's guide. https://peps.python.org/pep-0350/, 2023.

[37] Zhaoqiang Guo, Shiran Liu, Jinping Liu, Yanhui Li, Lin Chen, Hongmin Lu, and Yuming Zhou. How far have we progressed in identifying self-admitted technical debts? a comprehensive empirical study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(4):1–56, 2021.

[38] Cong Chen, Kang Zhang, and Takayuki Itoh. Empirical evidence of tags supporting high-level awareness. In *Cooperative Design, Visualization, and Engineering: 9th International Conference, CDVE 2012, Osaka, Japan, September 2-5, 2012. Proceedings 9*, pages 94–101. Springer, 2012.

[39] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. Online defect prediction for imbalanced data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 99–108. IEEE, 2015.

[40] Naufal Azmi Verdikha, Teguh Bharata Adji, and Adhistya Erna Permanasari. Study of undersampling method: Instance hardness threshold with various estimators for hate speech classification. *IJITEE (International Journal of Information Technology and Electrical Engineering)*, 2(2):39–44, 2018.

[41] Supatsara Wattanakriengkrai, Napat Srisermphoak, Sahawat Sintoplertchaikul, Morakot Choetkiertikul, Chaiyong Ragkhitwetsagul, Thanwadee Sunetnanta, Hideaki Hata, and Kenichi Matsumoto. Automatic classifying self-admitted technical debt using n-gram idf. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pages 316–322. IEEE, 2019.

[42] Kwabena Ebo Bennin, Jacky W Keung, and Akito Monden. On the relative value of data resampling approaches for software defect prediction. *Empirical Software Engineering*, 24(2):602–636, 2019.

[43] Alexander LeClair and Collin McMillan. Recommendations for datasets for source code summarization. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 3931–3937, 2019.

[44] Xin Xia, Emad Shihab, Yasutaka Kamei, David Lo, and Xinyu Wang. Predicting crashing releases of mobile applications. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–10, 2016.

[45] Zhong Li, Minxue Pan, Yu Pei, Tian Zhang, Linzhang Wang, and Xuandong Li. Empirically revisiting and enhancing automatic classification of bug and non-bug issues. *Frontiers of Computer Science*, 18(5):1–20, 2024.

[46] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[47] Yue Jiang, Bojan Cukic, and Yan Ma. Techniques for evaluating fault prediction models. *Empirical Software Engineering*, 13(5):561–595, 2008.

[48] Chris Mills, Gabriele Bavota, Sonia Haiduc, Rocco Oliveto, Andrian Marcus, and Andrea De Lucia. Predicting query quality for applications of text retrieval to software engineering tasks. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 26(1):1–45, 2017.

[49] Morakot Choetkiertikul, Hoa Khanh Dam, Truyen Tran, and Aditya Ghose. Predicting delays in software projects using networked classification (t). In *2015 30th IEEE/ACM international conference on automated software engineering (ASE)*, pages 353–364. IEEE, 2015.

[50] Ilya Loshchilov and Frank Hutter. Fixing weight decay regularization in adam. 2018.

[51] Jaechang Nam and Sunghun Kim. Clami: Defect prediction on unlabeled datasets (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 452–463. IEEE, 2015.

[52] Yibiao Yang, Yuming Zhou, Jinping Liu, Yangyang Zhao, Hongmin Lu, Lei Xu, Baowen Xu, and Hareton Leung. Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, pages 157–168, 2016.

[53] Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, 34(4):485–496, 2008.

[54] Andrew P Bradley. The use of the area under the roc curve in the evaluation of machine learning algorithms. *Pattern recognition*, 30(7):1145–1159, 1997.

[55] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2012.

[56] Tian Jiang, Lin Tan, and Sunghun Kim. Personalized defect prediction. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 279–289. Ieee, 2013.

[57] Kawin Ethayarajh. How contextual are contextualized word representations? comparing the geometry of bert, elmo, and gpt-2 embeddings. *arXiv preprint arXiv:1909.00512*, 2019.

[58] Anjan Karmakar and Romain Robbes. Inspect: Intrinsic and systematic probing evaluation for code transformers. *IEEE Transactions on Software Engineering*, 2023.

[59] Hongxin Li, Jingran Su, Yuntao Chen, Qing Li, and ZHAO-XIANG ZHANG. Sheetcopilot: Bringing software productivity to the next level through large language models. *Advances in Neural Information Processing Systems*, 36, 2024.

[60] Zhiyuan Wan, Lingfeng Bao, Debin Gao, Eran Toch, Xin Xia, Tamir Mendel, and David Lo. Appmod: Helping older adults manage mobile security with online social help. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 3(4):1–22, 2019.

[61] Franki YH Kung, Navio Kwok, and Douglas J Brown. Are attention check questions a threat to scale validity? *Applied Psychology*, 67(2):264–283, 2018.

[62] Emerson Murphy-Hill, Ciera Jaspan, Caitlin Sadowski, David Shepherd, Michael Phillips, Collin Winter, Andrea Knight, Edward Smith, and Matthew Jorde. What predicts software developers' productivity? *IEEE Transactions on Software Engineering*, 47(3):582–594, 2019.

[63] Anastasia Danilova, Alena Naiakshina, Stefan Horstmann, and Matthew Smith. Do you really code? designing and evaluating screening questions for online surveys with programmers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 537–548. IEEE, 2021.

[64] Ita Ryan, Utz Roedig, and Klaas-Jan Stol. Measuring secure coding practice and culture: A finger pointing at the moon is not the moon. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1622–1634. IEEE, 2023.

[65] Henry B Mann and Donald R Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.

[66] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, and Haofen Wang. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*, 2023.

[67] Zhengbao Jiang, Frank F Xu, Luyu Gao, Zhiqing Sun, Qian Liu, Jane Dwivedi-Yu, Yiming Yang, Jamie Callan, and Graham Neubig. Active retrieval augmented generation. *arXiv preprint arXiv:2305.06983*, 2023.

[68] Noor Nashid, Mifta Sintaha, and Ali Mesbah. Retrieval-based prompt selection for code-related few-shot learning. In *Proceedings of the 45th International Conference on Software Engineering (ICSE'23)*, 2023.

[69] The replicate package. scikit-learn. https://scikit-learn.org/stable/, 2021.

[70] Nathan Cassee, Fiorella Zampetti, Nicole Novielli, Alexander Serebrenik, and Massimiliano Di Penta. Self-admitted technical debt and comments' polarity: an empirical study. *Empirical Software Engineering*, 27(6):139, 2022.

[71] Fiorella Zampetti, Gianmarco Fucci, Alexander Serebrenik, and Massimiliano Di Penta. Self-admitted technical debt practices: a comparison between industry and open-source. *Empirical Software Engineering*, 26:1–32, 2021.

[72] Chao Ni, Xin Xia, David Lo, Xiang Chen, and Qing Gu. Revisiting supervised and unsupervised methods for effort-aware cross-project defect prediction. *IEEE Transactions on Software Engineering*, 2020.

[73] Haoye Wang, Zhipeng Gao, Tingting Bi, John Grundy, Xinyu Wang, Minghui Wu, and Xiaohu Yang. What makes a good todo comment? *ACM Trans. Softw. Eng. Methodol.*, 2024.

[74] Yasutaka Kamei, Everton da S Maldonado, Emad Shihab, and Naoyasu Ubayashi. Using analytics to quantify interest of self-admitted technical debt. In *QuASoQ/TDA@ APSEC*, pages 68–71, 2016.

[75] Nico Zazworka, Michele A Shaw, Forrest Shull, and Carolyn Seaman. Investigating the impact of design debt on software quality. In *Proceedings of the 2nd Workshop on Managing Technical Debt*, pages 17–23, 2011.

[76] Robert L Nord, Ipek Ozkaya, Philippe Kruchten, and Marco Gonzalez-Rojas. In search of a metric for managing architectural technical debt. In *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, pages 91–100. IEEE, 2012.

[77] Nico Zazworka, Rodrigo O Spínola, Antonio Vetro', Forrest Shull, and Carolyn Seaman. A case study on effectively identifying technical debt. In *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, pages 42–47, 2013.

[78] Everton da S Maldonado and Emad Shihab. Detecting and quantifying different types of self-admitted technical debt. In *2015 IEEE 7Th international workshop on managing technical debt (MTD)*, pages 9–15. IEEE, 2015.

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

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

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

[82] Gianmarco Fucci, Fiorella Zampetti, Alexander Serebrenik, and Massimiliano Di Penta. Who (self) admits technical debt? In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 672–676. IEEE, 2020.

[83] Laerte Xavier, Fabio Ferreira, Rodrigo Brito, and Marco Tulio Valente. Beyond the code: Mining self-admitted technical debt in issue tracker systems. In *Proceedings of the 17th international conference on mining software repositories*, pages 137–146, 2020.

[84] Christoffer Rosen, Ben Grawi, and Emad Shihab. Commit guru: analytics and risk prediction of software commits. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, pages 966–969, 2015.

[85] Zhongxin Liu, Xin Xia, Christoph Treude, David Lo, and Shanping Li. Automatic generation of pull request descriptions. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 176–188. IEEE, 2019.

[86] Meng Yan, Xin Xia, Yuanrui Fan, Ahmed E Hassan, David Lo, and Shanping Li. Just-in-time defect identification and localization: A two-phase framework. *IEEE Transactions on Software Engineering*, 2020.