Automating TODO-missed Methods Detection and Patching

ZHIPENG GAO, Shanghai Institute for Advanced Study of Zhejiang University, Shanghai, China
YANQI SU, Australian National University, Canberra, Australia
XING HU, Zhejiang University, Hangzhou, China
XIN XIA, Huawei, Hangzhou, China

TODO comments are widely used by developers to remind themselves or others about incomplete tasks. In other words, TODO comments are usually associated with temporary or suboptimal solutions. In practice, all the equivalent suboptimal implementations should be updated (e.g., adding TODOs) simultaneously. However, due to various reasons (e.g., time constraints or carelessness), developers may forget or even are unaware of adding TODO comments to all necessary places, which results in the *TODO-missed methods*. These "hidden" suboptimal implementations in *TODO-missed methods* may hurt the software quality and maintainability in the long-term. Therefore, in this article, we propose the novel task of *TODO-missed methods* detection and patching and develop a novel model, namely TODO-comment Patcher (TDPatcher), to automatically patch TODO comments to the *TODO-missed methods* in software projects. Our model has two main stages: offline learning and online inference. During the offline learning stage, TDPatcher employs the GraphCodeBERT and contrastive learning for encoding the TODO comment (natural language) and its suboptimal implementation (code fragment) into vector representations. For the online inference stage, we can identify the *TODO-missed methods* and further determine their patching position by leveraging the offline trained model. We built our dataset by collecting *TODO-introduced methods* from the top-10,000 Python GitHub repositories and evaluated TDPatcher on them. Extensive experimental results show the promising performance of our model over a set of benchmarks. We further conduct an in-the-wild evaluation that successfully detects 26 *TODO-missed methods* from 50 GitHub repositories.

CCS Concepts: • **Software and its engineering → Software evolution**; **Maintaining software;**

Additional Key Words and Phrases: TODO comment, SATD, technical debt, software inconsistency, contrastive learning

ACM Reference Format:
Zhipeng Gao, Yanqi Su, Xing Hu, and Xin Xia. 2024. Automating TODO-missed Methods Detection and Patching. *ACM Trans. Softw. Eng. Methodol.* 00, JA, Article 00 (November 2024), 28 pages. https://doi.org/10.1145/3700793

This research is supported by the Starry Night Science Fund of Zhejiang University Shanghai Institute for Advanced Study, Grant No. SN-ZJU-SIAS-001. This research is partially supported by the Shanghai Sailing Program (23YF1446900) and the National Science Foundation of China (No. 62202341). This research is partially supported by the Ningbo Natural Science Foundation (No. 2023J292).
Authors' Contact Information: Zhipeng Gao, Shanghai Institute for Advanced Study of Zhejiang University, Shanghai, China; e-mail: zhipeng.gao@xju.edu.cn; Yanqi Su, Australian National University, Canberra, Australia; e-mail: Yanqi.Su@anu.edu.au; Xing Hu (Corresponding author), Zhejiang University, Hangzhou, China; e-mail: xinghu@zju.edu.cn; Xin Xia, Huawei, Hangzhou, China; e-mail: xin.xia@acm.org.

ACM 1049-331X/2024/11-ART00
https://doi.org/10.1145/3700793

1 Introduction

TODO comments are widely used by developers to describe valuable code changes that can improve software quality and maintenance. In other words, the TODO comments pinpoint the current suboptimal solutions that developers should pay attention to in future development. For example, as shown in Example 1 of Figure 1, when a developer implemented the `save_workbook` method, they added a TODO comment (Line 456, *TODO: replace with path transformation functions*) to remind themselves or other developers for indicating the temporary path implementations (i.e., Line 457). Ideally, once a TODO comment is added for a method, all the methods with equivalent suboptimal implementations should be updated (i.e., adding the same TODO comment) correspondingly. Example 2 in Figure 1 shows such a situation, the method `get_fullname` contains similar suboptimal implementations (i.e., Line 493) as method `save_workbook`. Therefore, the developers added a TODO comment to the above two methods simultaneously in one commit. However, due to the complexity of the ever-increasing software systems (e.g., hundreds of files with thousands of methods) and frequent refactoring of the source code (e.g., taking over code from others), developers may forget or even are unaware of such suboptimal implementations somewhere else in the project, resulting in *TODO-missed methods*.

In our work, the definition of *TODO-missed method* is as follows: Given a method pair $\langle \mathbf{m}_1, \mathbf{m}_2 \rangle$ that has equivalent suboptimal implementations **I**, if $\mathbf{m}_1$ was added with a TODO comment **T** to indicate **I** but $\mathbf{m}_2$ was not, then we then define $\mathbf{m}_2$ as a *TODO-missed method* and $\mathbf{m}_1$ as a *TODO-introduced method*. Methods with proper TODO comments can help developers understand the source code context and prevent introducing mistakes when touching unfamiliar code. On the contrary, the *TODO-missed methods* miss the TODO comment specification, resulting in the suboptimal implementations being ignored (for a long time) or never getting revisited unless causing any damages. Take the above methods as an example: If the TODO comment is not specified clearly in `get_fullname`, then the developers can easily forget this unfinished task, and the problem is even worsened as software constantly evolves and developers frequently join and leave the development team. It is thus helpful to notify developers of *TODO-missed methods* in software projects before any unwanted side effects are caused.

In this article, we aim to identify *TODO-missed methods* when new TODO comments are added. Manually checking *TODO-missed methods* is a time-consuming and error-prone process, this is especially true for large software projects, usually with hundreds of files and thousands of lines of code. It would cost a great amount of effort if developers inspect all methods line by line to identify *TODO-missed methods*. Therefore, it is beneficial and desirable to have a tool for automating the checking of *TODO-missed methods* in practice. To perform this task, the following two key issues need to be handled properly:

(1) *TODO-missed methods* detection. Identifying *TODO-missed methods* first requires understanding the semantics of the TODO comment and its suboptimal implementation and the relationship between them. The comment and source code are of different types (i.e., free-form natural languages and formal programming languages), which causes naturally a gap between them. As shown in Figure 1 examples, TODO comments and their suboptimal implementations are semantically related but lexically isolated. As a result, it is a non-trivial task to determine whether a method should include a particular TODO comment or not. Therefore, an effective tool for detecting *TODO-missed methods* requires capturing the semantic mapping and implicit connections between the TODO comment and its corresponding suboptimal implementation(s).
(2) *TODO-missed methods* patching. After a *TODO-missed method* is detected, our next step aims to locate the exact suboptimal positions (e.g., the source code lines) that the TODO comment

COMMIT: 922096788127a1271b5c2b4e1544447c65b0f954

Ex1: *TODO-introduced* method: xlwings/xlwings/_xlmac.py

```
455    def save_workbook(xl_workbook, path):
456        # TODO: replace with path transformation functions
457        saved_path = xl_workbook.properties().get(kw.path)
458        if  (saved_path != '') and (path is None):
459            # Previously saved: Save under existing name
460            xl_workbook.save()
 …                    …
```

Ex2: *TODO-introduced* method: xlwings/xlwings/_xlmac.py

```
491    def get_fullname(xl_workbook):
492        # TODO: replace with path transformation functions
493        hfs_path = xl_workbook.properties().get(kw.full_name)
494        if hfs_path == xl_workbook.properties().get(kw.name):
495            return hfs_path
496        url = mactypes.covertpathtourl(hfs_path, 1)
497        return covertpathtourl(url, 1)
…                    …
```

Fig. 1. Examples of TODO-introduced methods.

should be patched to, namely *TODO-missed methods* patching. Patching TODO comments to the right location is also a non-trivial task with respect to the following reasons: It is very difficult, if not possible, to determine where a TODO comment should be added by just reading the method source code. For example, as shown in Example 2 in Figure 1, multiple statements within `get_fullname` method are associated with "path" actions, one cannot easily claim which code line is connected to the TODO comment. A more intuitive and reliable way is to take the *TODO-introduced method* as a reference and check the equivalent suboptimal positions by pairwise comparison.

To alleviate the above issues and help developers better maintain TODO comments, in this work, we propose a novel framework named (**TODO-comment Patcher (**TDPatcher**)** to automatically patch the TODO comments to the *TODO-missed methods*. The main idea of our approach is twofold: (i) *Code Embeddings*: Code patterns (e.g., the suboptimal implementations), as well as code documentation (e.g., TODO comments), can be automatically encoded into contextualized semantic vectors via the techniques adapted from the large-scale pre-trained models (e.g., GraphCodeBERT). (ii) *Contrastive Learning*: Learning semantic representation for an individual method alone is not sufficient, it is necessary to explore the correlations and implicit differences between the *TODO-introduced methods* and *TODO-missed methods*. In this work, we adopt the contrastive learning strategy to teach the model to pull positive samples together (i.e., methods with equivalent suboptimal implementations) while simultaneously pushing negative samples (i.e., irrelevant methods) apart.

TDPatcher consists of two stages: offline learning and online inference. During offline learning, we collect *TODO-introduced methods* (a method that newly adds TODO comment) from the Top-10,000 Python GitHub repositories. We then automatically establish $\langle anchor, positive, negative \rangle$ triplet training samples in terms of whether the *TODO-introduced method* can be paired. TDPatcher first employs the GraphCodeBERT [19] model to embed *anchor*, *positive*, and *negative* samples into contextualized vectors, respectively. TDPatcher then applies contrastive learning strategy [13, 21] to further learn the discriminative vector representations. The goal of our

framework is that equivalent suboptimal implementations (i.e., positive pairs) should be as close as possible in hidden vector space, while the irrelevant code patterns (i.e., negative pairs) should be as far away as possible in the space. When it comes to online prediction, for a given *TODO-introduced method*, TDPatcher first transforms its suboptimal code implementations into vectors, then locates the code fragments that have the nearest vectors to the suboptimal implementations.

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

(1) We propose a novel task of *TODO-missed methods* detection and patching, and we build a large dataset for checking *TODO-missed methods* from top-10,000 Python Github repositories. To the best of our knowledge, it is the first large dataset for this task.
(2) We propose a novel model, TDPatcher, to automatically detect and patch *TODO-missed methods* when TODO comments are introduced. TDPatcher can help developers to increase the quality and maintainability of software and alleviate the error-prone code review process.
(3) We extensively evaluate TDPatcher using real-world popular open source projects in Github. TDPatcher is shown to outperform several baselines and reduce the developer's efforts in maintaining the TODO comments.
(4) We have released our replication package, including the dataset and the source code of TDPatcher [1], to facilitate other researchers and practitioners to repeat our work and verify their ideas.

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

2 Motivation

In this section, we first show the motivating examples from real-world *TODO-missed methods*. We then present user scenarios that employ our proposed TDPatcher to address these problems.

2.1 Motivating Examples

Due to the large scale of modern software projects, developers may forget or even be unaware of adding TODO comments to the methods that are supposed to, leading to the introduction of *TODO-missed methods*. Figures 2 and 3 show two examples of *TODO-missed methods* in real-world GitHub repositories. From these examples we can see the following:

(1) *TODO-missed methods* can decrease the software quality and maintainability and induce potential problems and/or bugs in subsequent development. An example is shown in Figure 2, the developer added a TODO comment to `yolo_predict_mnn` method, noting that "*TODO: currently MNN python binding have mem leak when creating MNN, ... so we convert input image to tuple*." To address this memory leak issue, the developer modified the `image_data` to a tuple. Unfortunately, the developer neglected the same suboptimal implementation in another method (i.e., `validate_yolo_model_mnn`), leading to the latter becoming a *TODO-missed method*. As a result, the same problem could be caused when the `validate_yolo_model_mnn` method is triggered. Upon manual inspection of the project's commit history, we found that the developer eventually added the missing TODO comment

Repo: keras-YOLOv3-model-set (629 Stars) ***TODO-introduced Commit***: f007ff1f6214d91e22bc51eda6f25e7048492022

Commit Message: support MNN model evaluation in eval.py

TODO-introduced method: eval.py —> yolo_predict_mnn

```
    def yolo_predict_mnn(interpreter, session, image, anchors, num_classes, ..., v5_decode):
        ...
        # use a temp tensor to copy data
+       # TODO: currently MNN python binding have mem leak when creating MNN. Tensor
+       # from numpy array, only from tuple is good, so we convert input image to tuple
+       input_elementsize = reduce(mul, input_shape)
        tmp_input = MNN.Tensor( input_shape, input_tensor.getDataType(), \
-                               image_data, input_tensor.getDimensionType())
+                               tuple(image_data.reshape(input_elementsize, -1),  input_tensor.getDimensionType())
        ...
```

TODO-dropped method: validate_yolo_mnn.py —> validate_yolo_model_mnn

```
    def validate_yolo_model_mnn(model_path, image_file, anchors, class_names, loop_count):
        ...
        # use a temp tensor to copy data
        tmp_input = MNN.Tensor( input_shape, input_tensor.getDataType(), \
                                image_data, input_tensor.getDimensionType())
        ...
```

Repo: keras-YOLOv3-model-set (629 Stars) ***TODO-patching Commit***: bbd7a393546399f5dbaf19e12e4b1079726bb6a8

Commit Message: fix MNN input tensor copy issue

```
    def validate_yolo_model_mnn(model_path, image_file, anchors, class_names, loop_count):
        ...
-       # use a temp tensor to copy data
-       tmp_input = MNN.Tensor( input_shape, input_tensor.getDataType(), \
-                               image_data, input_tensor.getDimensionType())
+       # create a temp tensor to copy data
+       # use TF NHWC layout to align with image data array
+       # TODO: currently MNN python binding have mem leak when creating MNN. Tensor
+       # from numpy array, only from tuple is good, so we convert input image to tuple
+       tmp_input_shape = (batch, height, width, channel)
+       input_elementsize = reduce(mul, tmp_input_shape)
+       tmp_input = MNN.Tensor( tmp_input_shape, input_tensor.getDataType(), \
+                               tuple(image_data.reshape(input_elementsize, -1),  MNN.Tensor_DimensionType_Tensorflow)
        ...
```

Fig. 2. Motivating Example 1.

back to `validate_yolo_model_mnn` and fixed the problem within the *TODO-missed method* using the same way as the *TODO-introduced method*. The commit message ("*fix MNN input tensor copy issue*") further confirms our assumption that the problem was caused by overlooking the suboptimal implementation within the *TODO-missed method*.

(2) Identifying *TODO-missed methods* is a non-trivial task, given diverse forms of equivalent suboptimal implementations. Figure 3 shows another example of *TODO-missed method* in the `galaxy` project. As can be seen, the *TODO-introduced method* (i.e., `_check_files`) and the *TODO-missed method* (i.e., `get_current_branch`) have the same encoding problem related to the `subprocess` output. Despite sharing the same problem, the suboptimal code patterns within the *TODO-introduced method* (i.e., `yield line.strip()`) and the *TODO-missed method* (i.e., `return subprocess.check_out().strip()`) differ significantly. The developer can easily ignore such suboptimal code implementations due to time constraints and/or carelessness, and has to spend extra time and effort for debugging when problems are eventually exposed. Regarding this particular case, the developer fixed the encoding issue in the *TODO-missed method* after a week when the TODO comment was initially added. During this time, other team members may revise the code without noticing the suboptimal code patterns in the source code, therefore any code refactoring related to this *TODO-missed method* can be influenced, posing potential threats to software maintenance and evolution.

In summary, the *TODO-missed methods* can be easily ignored by developers and introduce potential problems and/or bugs in subsequent development. Therefore, we propose TDPatcher in this work, which aims to make the hidden suboptimal implementations *observable* by adding TODO

Repo: galaxy (839 Stars) ***TODO-introduced Commit***: 4c9280c2c782ffd98828bd3a67f9f44502765d6c

Commit Message: Decode output of external process calls. Since Python 3.x subprocess use byte streams by default for stdout and stderr, … to keep compatibility with Python 2 wu use `.decode('utf-8')` instead, until Python 2 support is dropped.

TODO-introduced method: __init__.py —> _check_files

```
  def _check_files(self, paths):
      …
      proc = subprocess.Popen(cmd, cwd=self.root, stdout=subprocess.PIPE)
      for line in proc.stdout:
-         yield line.strip()
+         # TODO(cutwater): Replace `.decode('utf-8')` call with subprocess
+         # parameter `encoding` after dropping Python 2.7 support.
+         yield line.decode('utf-8').strip()
      proc.wait()
```

TODO-dropped method: git.py —> get_current_branch

```
  def get_current_branch(directory=None):
      …
      cmd = ['git', 'rev-parse', '—abbrev-ref', 'HEAD']
      return subprocess.check_out(cmd, cwd=directory).strip()
```

Repo: galaxy (839 Stars) ***TODO-patching Commit***: c49e3080fbd5f41fae78e8e31bd4d670ef91a104

Commit Message: Python 3: Fix subprocess output encoding

```
  def get_current_branch(directory=None):
      …
      cmd = ['git', 'rev-parse', '—abbrev-ref', 'HEAD']
-     return subprocess.check_out(cmd, cwd=directory)..strip()
+     # TODO(cutwater): Replace `.decode('utf-8')` call with subprocess
+     # parameter `encoding` after dropping Python 2.7 support.
      return subprocess.check_out(cmd, cwd=directory).decode('utf-8').strip()
      …
```

Fig. 3. Motivating Example 2.

comments to the *TODO-missed methods*. With the help of TDPatcher, developers will be reminded of the ignored suboptimal implementations just-in-time, such immediate feedback ensures the context is still fresh in the minds of developers. This fresh context can help developers to be aware of their suboptimal code and thus avoid the introduction of the *TODO-missed methods*.

2.2 User Scenarios

We illustrate a usage scenario of TDPatcher as follows.

Without Our Tool: Consider Bob is a developer. One day, when Bob implemented a method to add a new feature, he realized that the current solution was suboptimal. He then adds a TODO comment to indicate the features that are currently not supported or the optimizations that need to be implemented. Besides the current method, there are multiple methods with the equivalent suboptimal implementation that are supposed to be handled correspondingly. However, due to time constraints or unfamiliarity with the software system, Bob is not aware of other suboptimal positions and thus introduces the *TODO-missed methods* unconsciously. These *TODO-missed methods* may negatively impact program quality and maintenance. Furthermore, when other developers take over the code from Bob, they have no idea that the method is suboptimal, the new updates on this method are risky and may introduce bugs in the future.

With Our Tool: Now consider Bob adopts our tool, TDPatcher. After introducing TODO comments and their suboptimal solutions, Bob can use our tool to automatically identify the presence of the *TODO-missed methods*, and such immediate feedback can ensure the code context is still fresh in the developer's mind. Moreover, TDPatcher can locate the specific source code line where the TODO comments are supposed to be added, thus helping developers to locate the suboptimal positions with less inspection effort. With the help of our tool, Bob successfully finds and patches the *TODO-missed methods* in the current software repository efficiently, which increases the reliability and maintenance of the system and decreases the likelihood of introducing bugs.

3 Approach

In this section, we first define the task of *TODO-missed methods* detection and patching for our study. We then present the details of our data preparation and proposed approach.

3.1 Task Definition

Our tool aims to automatically detect and patch *TODO-missed methods* in software projects. Formally, given a method pair $\langle \mathbf{m_1}, \mathbf{m_2} \rangle$, let $\mathbf{m_1}$ be the *TODO-introduced method* and $\mathbf{T}$ be the TODO comment associated with $\mathbf{m_1}$, and let $\mathbf{m_2}$ be the candidate method to be checked. Our first task is to find a function `detect` so that

$$detect(\mathbf{m_1}, \mathbf{m_2}, \mathbf{T}) = \begin{cases} 1 & \text{for} \quad \mathbf{m_1} \rightleftharpoons \mathbf{m_2} \\ 0 & \text{for} \quad otherwise \end{cases}, \tag{1}$$

where $\mathbf{m_1} \rightleftharpoons \mathbf{m_2}$ denotes that the m_1 and m_2 contains the equivalent suboptimal implementations. Our second task is to add the TODO comment $\mathbf{T}$ to the right position of $\mathbf{m_2}$ if $detect(\mathbf{m_1}, \mathbf{m_2}, \mathbf{T}) = 1$. Let $\mathbf{p}$ be the desired adding position of $\mathbf{T}$ in $\mathbf{m_2}$. Our second task is to find a function `patch` so that

$$patch(\mathbf{m_1}, \mathbf{m_2}, \mathbf{T}) = \mathbf{p}. \tag{2}$$

3.2 Data Collection

We first present the details of our data collection process. We build our dataset by collecting data from the top 10,000 Python repositories (ordered by the number of stars) in GitHub until December 25, 2022. We have to mention that while this study utilizes Python for investigation, our approach could be applied to source code written in any programming language.

3.2.1 Identifying TODO-introducing Commits. We first collected and cloned the top-10K Python repositories from GitHub until December 2022. The git repository stores software update history, each update comprises a `diff` file that shows the differences between the current and previous versions of the affected files. For each cloned repository, we first checkout all the commits from the repository history. Following that, for each commit, if "TODO" appears within *added lines* of the `diff`, then we consider this commit as a *TODO-introducing commit*. Among the top-10K Python repositories, we discovered that 6,519 of them contained *TODO-introducing commits*. Finally, we have collected 148,675 *TODO-introducing commits* (including 196,945 TODOs, since there are commits containing more than one TODO) from these 6,519 Python repositories. Considering the forks of an original repository could introduce noise data samples and contaminate the training and testing dateset, we further checked that all 6,519 Python repositories are original repositories and not forks from other repositories.

3.2.2 Cleaning TODO-introducing Commits. Although we have gathered *TODO-introducing commits* from Python repositories, it is not guaranteed that all TODO comments are added to Python methods. In this step, we only preserve *TODO-introducing commits* if they contain TODO comments added to Python source files and remove any other unrelated commits. To achieve this, we apply `Pydriller` [51] (a framework for analyzing Git repositories) to extract modified files of each *TODO-introducing commit*, and only retain commits when their associated modified files are Python source files (ending up with `.py`). As a result, we retain a total of 112,951 verified *TODO-introducing commits* (including 152,724 TODOs) from 5,888 Python repositories.

3.2.3 Extracting TODO-introduced Methods. In this step, We extract methods that are associated with the TODO comments. Particularly, for a given verified *TODO-introducing commit*, we first check out affected files from this commit. We then pinpoint the line number of the TODO comment and automatically extract the *TODO-introduced method* if the method's code range covers the line

of the TODO comment. During this process, we apply the following rules: *(i) Rule1: The TODO comments that consist of less than three words are excluded.* These TODOs are often too short to carry meaningful code indications. As a result, we removed 9,660 *TODO-introducing commits* (including 15,957 TODOs), leaving us with 136,767 TODOs for this step. *(ii) Rule2: The TODO comments that are not located within a method are excluded.* Since our focus is on addressing *TODO-missed methods* in this study, we disregard TODO comments that are not associated with any methods. By doing so, we removed 39,354 TODOs that were outside of methods and retained 97,413 within method TODOs. *(iii) Rule3: The methods that failed to parse are excluded.* To extract method-level information for downstream tasks, we use Python 2.7/3.7 standard library `ast` to produce the file-level **Abstract Syntax Tree (AST)** for Python 2 and Python 3 projects, respectively. We then extract every individual method and class method source code via inspecting the method related AST node (e.g., the `FunctionDef` and `AsyncFunctionDef`). We use the `autopep8` to overcome the issues of different styles and white space or tab conventions. Finally, 1,451 methods failed to parse due to syntax error, 95,962 *TODO-introduced methods* are successfully parsed and extracted for downstream tasks.

3.2.4 Grouping TODO-introduced Methods. This step is responsible for grouping methods with the same TODO comments together. Different methods may contain the same TODO comments, which indicates the same suboptimal code implementations. In this step, for a given project, we automatically check each *TODO-introduced method* within this project. Methods with the same TODO comment will be grouped together, while methods with unique TODO comment will be excluded. As a result, each TODO comment will be paired with a set of methods (e.g., different methods or the same method of different TODO locations). Finally, the *TODO-introduced method* groups, $\mathcal{G} = \{(\mathbf{T_i}, \mathbf{m_{i_1}}, \mathbf{m_{i_2}}, \ldots, \mathbf{m_{i_n}})\}_{i=1}^{g}$ are constructed, where $\mathbf{m_{i_1}}, \ldots, \mathbf{m_{i_n}}$ are a group of methods with the same TODO comment $\mathbf{T_i}$. Each group of methods is regarded as methods with equivalent suboptimal code implementations. Among the 95,962 *TODO-introduced methods*, 18,126 of them are grouped into 6,855 *TODO-introduced method* groups, while the other 77,386 *TODO-introduced methods* are excluded.

3.2.5 Manual Validation. Since we automatically built our dataset of *TODO-introduced methods* from Top-10K Python repositories, we cannot ensure that there are no outlier cases, or noisy data, during the data preparation process. Therefore, we further performed a manual checking step to validate the quality of our constructed dataset. Specifically, we randomly sampled 100 *TODO-introduced method* groups from our dataset. Then the first author manually examined whether *TODO-introduced methods* within each group contains equivalent suboptimal implementations. Finally, 96 groups of methods are marked as methods with equivalent suboptimal implementations. Thus, we are confident with the quality of our constructed dataset.

3.2.6 Empirical Findings. We have gathered the following insights from our data collection process: (i) *TODOs are widely used by developers in open source projects.* As shown in Section 3.2.1, among the top-10K Python repositories, 6,519 of them incorporate TODOs during the software development. This confirms the significant role of TODO comments in managing and coordinating diverse programming tasks. (ii) *The majority of TODOs are added within methods.* As demonstrated in our preliminary investigation in Section 3.2.3, among the 136,767 introduced TODOs, approximately 71% (97,413 TODOs) are inserted within methods, which verifies the importance and necessity of targeting our objectives, i.e.,*TODO-missed methods.* (iii) We have identified 6,855 *TODO-introduced method* groups in Section 3.2.4. Ideally, methods within the same group should be updated (i.e., adding TODOs) simultaneously. However, after our investigation, 3,436 groups (i.e., 50.1%) of methods added TODOs within the same commit, while the remaining 3,419 (i.e., 49.9%)

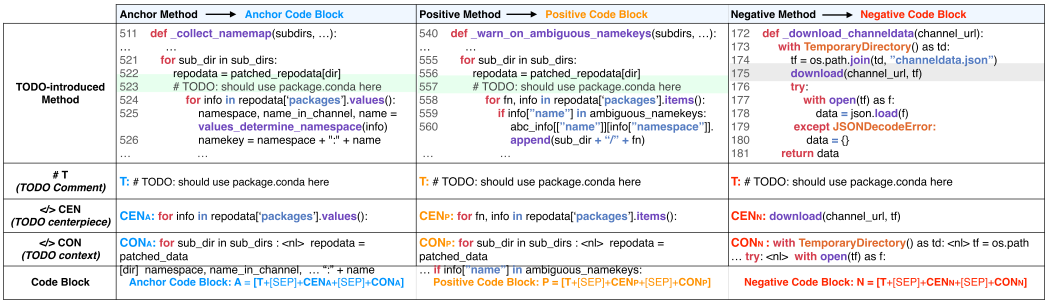

Fig. 4. Code block generation examples.

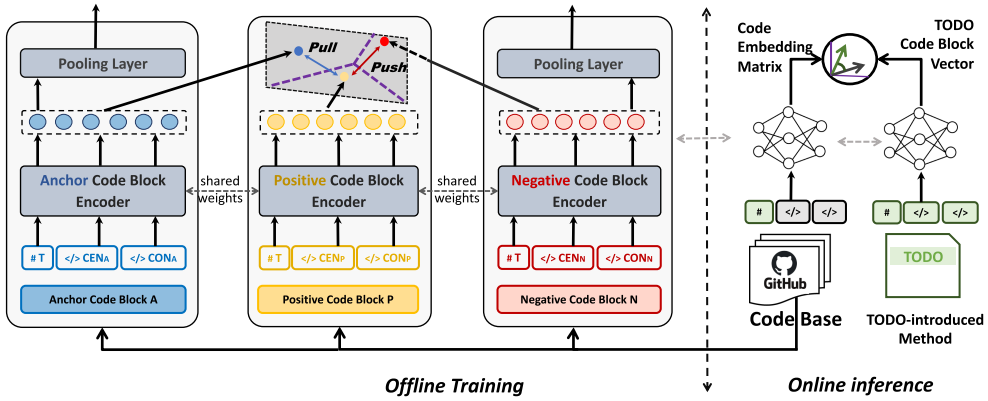

Fig. 5. Overall framework of our TDPatcher.

groups added TODOs across different commits. This phenomenon signals that *a large number of TODOs are not added in time by developers, leading to the introduction of TODO-missed methods.* It is thus highly desirable to have a tool that provides just-in-time automatic detection of *TODO-missed methods* and add them before they mislead developers and/or cause any unwanted side effects. This automated process of detecting and patching *TODO-missed methods* can substantially increase the quality and reliability of the software system.

3.3 Approach Details

In this section, we first present the details of constructing triplets model inputs, as shown in Figure 4. We then describe the overall framework of our approach, named TDPatcher, as illustrated in Figure 5. The approach details are as follows:

3.3.1 Model Inputs Construction. To capture semantic correlations between equivalent code implementations, we adopt the idea of contrastive learning. Different from traditional software engineering tasks such as defect prediction [55, 58], contrastive learning requires triplets as inputs [6, 13, 61]. Therefore, we need to prepare the data as triplets for model inputs. Each triplet is composed of three code blocks, namely *anchor code block* (denoted as **A**), *positive code block* (denoted as **P**), and *negative code block* (denoted as **N**), respectively. Each code block consists of three parts: **T**, **CEN**, and **CON**, where **T** represents the *TODO comment*, **CEN** represents the *TODO centrepiece*, and **CON** represents the *TODO context*, respectively. Figure 4 demonstrates the code block generation process. The detailed definitions of *TODO centrepiece* and *TODO context* are explained as follows:

— *Anchor Code Block Generation.* After data collection, we obtain the *TODO-introduced method* groups $\mathcal{G}$, for a given group $(\mathbf{T}, \mathbf{m_1}, \mathbf{m_2}, \ldots, \mathbf{m_n})$ in $\mathcal{G}$, we randomly select a method $\mathbf{m_k}(1 \leq k \leq n)$ as the anchor method. We construct the anchor code block for $\mathbf{m_k}$ as follows: we identify the code line following the TODO comment as the *TODO centrepiece*; We consider the two code lines before and after the TODO comment (excluding the *TODO centrepiece*) as the *TODO context*. Finally, the TODO comment, the *TODO centrepiece*, and the *TODO context* make an anchor code block. In summary, the anchor code block $\mathbf{A} = [\mathbf{T}, \mathbf{CEN_A}, \mathbf{CON_A}]$, where $\mathbf{CEN_A}$ and $\mathbf{CON_A}$ are the *TODO centrepiece* and *TODO context* of the anchor method.

— *Positive Code Block Generation.* The positive code block refers to the code fragment that shares equivalent suboptimal implementations with the anchor code block. For a given *TODO-introduced method* group $(\mathbf{T}, \mathbf{m_1}, \ldots, \mathbf{m_n})$, once the method $\mathbf{m_k}$ is selected as the anchor method, each of the rest methods $\mathbf{m_j}(1 \leq j \leq n, j \neq k)$ will serve as a candidate positive method. Given the positive method $\mathbf{m_j}$, we generate the positive code block exactly the same as making the anchor code block. Similarly, the positive code block $\mathbf{P} = [\mathbf{T}, \mathbf{CEN_P}, \mathbf{CON_P}]$ can be constructed, where $\mathbf{CEN_P}$ and $\mathbf{CON_P}$ are the *TODO centrepiece* and *TODO context* of the positive method $\mathbf{m_j}$.

— *Negative Code Block Generation.* The negative code block refers to the irrelevant code fragment in terms of the anchor code block. The negative code block also includes three segments. The TODO comment is the same comment with the anchor/positive code block. Regarding the *TODO centrepiece*, we randomly select a source code line from a non-TODO method as the negative *TODO centrepiece*, the two code lines before and after the centrepiece are regarded as the negative *TODO context*. Likewise, a negative code block $\mathbf{N} = [\mathbf{T}, \mathbf{CEN_N}, \mathbf{CON_N}]$, where $\mathbf{CEN_N}$ and $\mathbf{CON_N}$ represents the negative TODO centrepiece and negative TODO context, respectively.

So far, given a *TODO-introduced method* group $(\mathbf{T}, \mathbf{m_1}, \mathbf{m_2}, \ldots, \mathbf{m_n})$, once the anchor method is determined, the remaining method within the group is considered a positive method. Then each positive method will be paired with the anchor method and a negative method to establish a triplet sample. In other words, the *TODO-introduced method* group $(\mathbf{T}, \mathbf{m_1}, \mathbf{m_2}, \ldots, \mathbf{m_n})$ can generate $n-1$ triplet samples. From each triplet sample, the anchor code block **A**, positive code block **P**, and negative code block **N** are extracted automatically for model inputs, denoted as $\langle \mathbf{A}, \mathbf{P}, \mathbf{N} \rangle$. The positive code block **P** paired with the anchor code block **A** makes a positive pair (i.e., $\langle \mathbf{A}, \mathbf{P} \rangle$), while the negative code block **N** paired with the anchor code block **A** makes a negative pair (i.e., $\langle \mathbf{A}, \mathbf{N} \rangle$). In this work, the positive pairs are code blocks with equivalent suboptimal implementations, while the negative pairs are code blocks with irrelevant implementations. Since our code block consists of three parts (i.e., *TODO comment*, *TODO centrepiece* and *TODO context*), we add a special token `[SEP]` between different parts to further separate the natural language (TODO comment) and code implementations (i.e., TODO centrepiece and TODO context), which has been proven to be effective for bridging the gap between heterogeneous data. Finally, we constructed 10,775 triplet samples from 6,855 *TODO-introduced method groups.*

3.3.2 Code Embedding Networks. Due to the lexical gap between the TODO comments (i.e., natural language) and their suboptimal implementations (i.e., source code), it is difficult for traditional embedding techniques such as Bag-of-words, TF-IDF, and word2vec to embed both source code and natural language as vectors. To capture the semantic correlation between the TODO comment and its suboptimal solutions, we employ the large-scale pretrained model, GraphCodeBERT [19], as the encoder component for our model. GraphCodeBERT has several advantages as compared with other code embedding techniques: First, different from traditional embedding techniques that produce context-independent embeddings, GraphCodeBERT generates contextual embeddings based

on local code implementations. Moreover, instead of regarding the code snippet as a sequence of tokens, GraphCodeBERT uses dataflow in pre-training stage, which captures the semantic-level code structure and encodes the intricate relationship between variables. GraphCodeBERT has been proven to be effective in many code-related tasks [19, 57].

Because our model inputs are triplets, TDPatcher adopts three encoders based on GraphCodeBERT, i.e., *Anchor Code-block Encoder*, *Positive Code-block Encoder*, and *Negative Code-block Encoder*. These three encoders share weights of a single GraphCodeBERT as shown in Figure 5. The GraphCodeBERT encoder contains 12 layers of transformers. The hidden size of each layer is 768. Each layer has a self-attention sub-layer that is composed of 12 attention heads. After feeding the code block sequence into the encoder, we extract the final hidden states $\mathbf{h}$ of the special token [CLS] as the code block embedding vector. As a result, the three encoders map triplet inputs, i.e., anchor code block **A**, positive code block **P**, negative code block **N**, into their corresponding code embeddings $\mathbf{h_A}$, $\mathbf{h_P}$, and $\mathbf{h_N}$, respectively.

3.3.3 Contrastive Learning. Now we present how to train TDPatcher to distinguish equivalent code implementations from irrelevant implementations. The goal of our contrastive learning is that the positive pairs (e.g., equivalent code blocks) should be as close as possible in higher-dimensional vector space, and the negative pairs should be as far away as possible in the space. In this step, we introduce a triplet network to further learn the discriminative features among different inputs. The triplet network is a contrastive loss function based on the cosine distance operator [10, 22, 61]. The purpose of this function is to minimize the distances between the embeddings of similar code blocks (i.e., $\mathbf{h_A}$ and $\mathbf{h_P}$) and maximize the distances between the embeddings of irrelevant code blocks (i.e., $\mathbf{h_A}$ and $\mathbf{h_N}$). Formally, the training objective is to minimize the following loss function:

$$max(||\mathbf{h_A} - \mathbf{h_P}|| - ||\mathbf{h_A} - \mathbf{h_N}|| + \epsilon, 0), \tag{3}$$

where ϵ is the margin of the distance between **A** and **N**. Intuitively, the triplet loss encourages the cosine similarity between two equivalent code implementations to go up, and the cosine similarity between two irrelevant code blocks to go down.

3.3.4 TODO-missed Methods Detection and Patching. Based on the code embedding and contrastive learning techniques, we are able to apply our approach to solve the *TODO-missed methods* detection and patching tasks.

- Regarding the *TODO-missed methods* detection task, for a codebase, we first extract all possible code blocks for each method, in this study, every consecutive five code lines is regarded as a code block (see the code block definition in Section 3.3.1). After that, we encode each code block into a vector by using our TDPatcher. When a new TODO comment is introduced, TDPatcher first computes the vector representation for the suboptimal code block regarding the TODO comment, we then search code blocks in our codebase that are "similar" to the suboptimal code fragment. Particularly, given vector representations of a code block and the suboptimal code fragment, if their cosine similarity score is over a pre-defined threshold, then we determine the code block as a suboptimal code block. The methods containing suboptimal code block(s) will be identified as *TODO-missed methods*. We set the predefined threshold to 0.9 in this study. The threshold is a key parameter that governs whether a method will be predicted as a *TODO-missed method*. Grid search is employed to select the optimal threshold between 0.6 and 1.0 with a step size of 0.01. We carefully tune the threshold on our validation set, the threshold to the best performance (i.e., F1-score) is selected as the optimal threshold (0.9 in this study) and used to evaluate our testing set.
- Regarding the *TODO-missed methods* patching task, we aim to find the exact patching position for the TODO comment. After identifying the *TODO-missed method*, we divide the

method into a list of code blocks (every consecutive five code lines) sequentially and choose the code block with the highest similarity score as the patching code block. The central code line of the patching code block (i.e., the third code line within each code block) is regarded as the patching position for the *TODO-missed method*.

4 Evaluations

In this section, we evaluate how well our approach for the task of *TODO-missed methods* detection and patching. In particular, we first introduce the experimental setup for our work, and we then aim to answer the following three key research questions:

— **RQ-1:** How effective is TDPatcher for *TODO-missed methods* detection?
— **RQ-2:** How effective is TDPatcher for *TODO-missed methods* patching?
— **RQ-3:** How effective is TDPatcher for using GraphCodeBERT and contrastive learning techniques?
— **RQ-4:** How effective is TDPatcher under different code block configurations?

4.1 Evaluation Setup

In this study, we constructed 10,775 triplet samples from 1,793 GitHub projects for training and evaluation. We split the constructed triplet samples into three chunks by projects: Eighty percent of projects' (i.e., 1,443 projects) triplet samples are used for training, 10% (i.e., 175 projects) are used for validation, and the rest (i.e., 175 projects) are held-out for testing. As a result, our final training, validation and test sets consist of 8,531, 1,114, and 1,130 triplet samples, respectively. It is worth mentioning that we split our dataset *projectwise* instead of *timewise*. In other words, we performed the cross-project evaluation settings [41, 47, 64, 71] for this study, which ensures our training samples, validation samples, and testing samples come from different projects. The training set was used to fit the model for our task, while the validation set was used to find and optimize the best model under different hyperparameters. The testing set was used for testing the final solution to confirm the actual predictive power of our model with optimal parameters.

4.2 RQ-1. How Effective Is TDPatcher for TODO-missed Methods Detection?

4.2.1 Experimental Setup. In this research question, we want to determine the effectiveness of our approach for identifying the *TODO-missed methods*. The testing set contains 1,130 triplet samples. Each triplet relates to an anchor method $\mathbf{m_a}$, a positive method $\mathbf{m_p}$, and a negative method $\mathbf{m_n}$. One way to evaluate our approach is to see whether it can correctly predict $\langle \mathbf{m_a}, \mathbf{m_p} \rangle$ as positive and $\langle \mathbf{m_a}, \mathbf{m_n} \rangle$ as negative. To make the testing environment similar to the real-world working environment, we construct a candidate method pool $\mathcal{P}$ for evaluation. Theoretically, $\mathcal{P}$ should involve all methods within this project. However, it is too expensive to do so due to the huge number of methods for large software projects. In our preliminary study, $\mathcal{P}$ is constructed by gathering all methods associated with the TODO-introduced files (more than 70% *TODO-introduced methods* are added within the same file). In other words, the candidate method pool $\mathcal{P}$ for testing anchor method $\mathbf{m_a}$ contains N methods, one of which is $\mathbf{m_p}$, where N is the total number of extracted methods. For evaluation, we pair $\mathbf{m_a}$ with each candidate method in $\mathcal{P}$ and feed them into TDPatcher to calculate a matching score; if the matching score is over a pre-defined threshold, then the candidate method is regarded as a *TODO-missed method*.

4.2.2 Evaluation Metrics and Baselines. The *TODO-missed methods* detection is a binary classification problem, we thus adopt the *Precision*, *Recall*, and *F1-score* to measure the performance of our approach. To demonstrate the effectiveness of our proposed model, TDPatcher, we compared it with the following chosen baselines:

— *Random Guess:* **Random-Guess (RG)** is usually selected as a baseline when the previous work is lacking [64]. For each candidate method, we randomly determine whether it is a *TODO-missed method* or not.
— *Centrepiece-Exact-Match:* **Centrepiece-Exact-Match (CEM)** is a heuristic baseline to identify *TODO-missed method.* Since the TODO centrepiece is closely related to the TODO comment, we compare the TODO centrepiece with each line of the candidate method; if any lines match the TODO centrepiece exactly, then we declare this candidate method as a *TODO-missed method.*
— *Centrepiece-Soft-Match:* **Centrepiece-Soft-Match (CSM)** modifies CEM by looking at the overlapping words between the TODO-centrepiece and the target line. If the text similarity (proportion of the overlapped words) is above a threshold, then we claim the candidate method as a *TODO-missed method.*
— *NiCad*: NiCad [48] is a widely used traditional clone detection tool that handles a range of programming languages (e.g., C, Java, Python, C#, PHP, Ruby, Swift) in terms of different granularities (i.e., method-level and block-level). Given two methods are code clones, these two methods are likely to introduce TODO comments at the same time. It is thus reasonable to adopt a clone detection method as a baseline. For a given a *TODO-introduced method*, the clone method will be regarded as its *TODO-missed method.* In this study, we thus adopt NiCad as a tool to detect code clones.
— ***Neural Code Search (NCS):*** Another thread of similar research that is relevant to our work is code search. Identifying the *TODO-missed method* can be viewed as searching code snippets related to *TODO-introduced method.* A plethora of approaches have been investigated for code search in software repositories [16, 18, 49, 70], and the key idea of these studies is mapping code to vectors and matching data in high-dimensional vector space. In this study, we choose NCS model proposed by Facebook [49] as the code search method baseline. NCS encoded the code to vectors by combining FastText embeddings and TF-IDF weightings derived from the code corpus. We can easily search for code "similar" to the *TODO-introduced* method in their vector space.
— *Large Pre-trained Models:* Recently, the large pre-trained models, such as the CodeBERT [12], PLBART [2], and CodeT5 [59], have proven to be effective for many downstream tasks, including code generation, code summarization, code translation, code clone detection [2, 26, 33, 54, 56, 65]. In this study, two large pre-trained models, i.e., CodeBERT and PLBART are chosen for our study. We use CodeBERT and PLBART to encode code and detect code clones on method-level, denoted as $\textbf{CodeBERT}_{\textbf{M}}$ and $\textbf{PLBART}_{\textbf{M}}$, respectively. Since TDPatcher detects the *TODO-missed methods* at the block-level (i.e., every consecutive five code lines of a method is regarded as a code block). For a fairer comparison, we also use CodeBERT and PLBART to detect code clones at the same block-level, denoted as $\textbf{CodeBERT}_{\textbf{B}}$ and $\textbf{PLBART}_{\textbf{B}}$. If any code blocks of a method are identified as code clones of a *TODO-introduced method*, then we claim this method as a *TODO-missed method.*

4.2.3 Experimental Results. The experimental results of our TDPatcher and baselines are summarized in Table 1. For each method mentioned above, the involved parameters were carefully tuned and the parameters with the best performance are used to report the final comparison results. From this table, we can observe the following points: RG achieves the worst performance regarding Precision, this is because our candidate method pool $\mathcal{P}$ has more than 16 methods on average, which shows the data imbalance problem in real scenarios. In general, the methods based on textual similarity, i.e., CEM and CSM, can achieve a relatively high precision score but low recall and F1 score. It is a little surprising that CEM achieves 65.2% precision by naively

Table 1. TODO-missed Methods Detection Evaluation

Approach	Precision	Recall	F1
RG	6.6%	49.7%	11.5%
CEM	65.2%	39.2%	49.0%
CSM	60.5%	46.1%	52.3%
NiCad	71.5%	35.4%	47.3%
NCS	55.3%	54.2%	54.8%
PLBART_{M}	49.3%	47.6%	48.4%
$\text{CodeBERT}_{\text{M}}$	36.8%	56.5%	44.6%
PLBART_{B}	70.2%	68.8%	69.5%
$\text{CodeBERT}_{\text{B}}$	73.1%	68.3%	70.6%
TDPatcher	83.7%	77.3%	80.4%

checking the exact match with the TODO centrepiece statement, which indicates the TODO comments are closely related to the TODO centrepiece statements. CSM performs better than CEM in terms of F1-score and Recall, this is because CSM modifies the exact matching strategy with the soft matching by allowing a small proportion of different tokens, which also causes a performance drop regarding the Precision score. However, both CEM and CSM are based on bag-of-words matching, which can only capture the lexical level features, while the suboptimal implementations of *TODO-missed methods* may be semantically related but lexically independent, which also explains their overall poor performance on Recall and F1 score.

The traditional clone-detection tool NiCad and code search tool NCS achieve a relatively high precision but low recall, which explains its overall unsatisfactory performance. Both NiCad and NCS can be viewed as variants of the problem of finding "similar" code. In other words, NiCad and NCS search for code in a code base "similar" to a given piece of code (i.e., *TODO-missed method*). The evaluation results show that even NiCad and NCS can accurately identify *TODO-missed methods*, but they miss a large number of actual positive instances. NCS outperforms NiCad in terms of all evaluation metrics, this is because NiCad also shares the disadvantage of CEM/CSM, it detects code clones based on text similarities, which is unable to capture the semantics of TODO comments. Compared with NiCad, NCS encodes the code snippets into vector representations by using traditional word embedding techniques (i.e., FastText), which verifies the effectiveness of the idea of using code embeddings for this study.

It is obvious that the large pre-trained models perform better on the block level (i.e., $\textbf{CodeBERT}_{\textbf{B}}$ and $\textbf{PLBART}_{\textbf{B}}$) than the method level (i.e., $\textbf{CodeBERT}_{\textbf{M}}$ and $\textbf{PLBART}_{\textbf{M}}$) by a large margin. Method-level approaches achieve a poor performance regarding the Precision and F1-score, the possible reason may be that the TODO comments are often associated with local suboptimal implementations, while the large pre-trained models encode the complete methods into vector representations, which brings a higher level of noise due to the very large size of method bodies. The block-level pre-trained models, i.e., $\textbf{PLBART}_{\textbf{B}}$ and $\textbf{CodeBERT}_{\textbf{B}}$, have their advantages as compared to the textual clone detection–based method (i.e., NiCad) and code search–based method (i.e., NCS). We attribute this to the following reasons: (i) First, compared with NiCad, which treats words as discrete symbols and ignores the words' semantics, the code semantic features can be automatically encoded into numerical vectors by pre-trained models. (ii) Second, compared with NCS, which uses traditional word embedding techniques, the large pre-trained models generate contextual embeddings. Specifically, the traditional word embedding techniques (e.g., word2vec, FastText) are context-independent, while the embeddings of pre-trained models are context-dependent. For example,

regarding NCS model, for a given `print` statement (e.g., `print('ip:'+internet.address)`), the same embeddings will be used for all instances of the same code pattern. However, even the commonly used `print` statement can serve as suboptimal implementations (e.g., *TODO: remove this debug statement before releasing*) depending on its context. In contrast, CodeBERT and PLBART generate different embeddings for the commonly used `print` statement based on its code context.

Our proposed model, TDPatcher, outperforms all the baseline methods by a large margin in terms of all evaluation metrics. We attribute the superiority of our model to the following reasons: (i) We employ GraphCodeBERT to encode the inputs (including TODO comment and local code implementations) into contextual vectors. Similarly to CodeBERT and PLBART, GraphCodeBERT takes advantage of large pre-trained models to generate contextual embeddings. (ii) GraphCodeBERT has its own advantage as compared to other large pre-trained models (e.g., CodeBERT and PLBART) by leveraging dataflow graphs to learn code representations. The use of dataflow in GraphCodeBERT brings the following benefits to our task: (a) Dataflow represents the dependency relation between variables; in other words, dataflow represents the relation of "where-the-value-comes-from" between variables. This code structure provides crucial code semantic information for code understanding [19], especially in the context of our local code implementations. For example, as illustrated in `_check_files()` in Figure 3, linking the suboptimal variable `line` with `subprocess` directly is challenging, dataflow provides a better way to understand the variable `line` comes from `proc`, while the value of `proc` comes from `subprocess`. (b) Compared with AST, dataflow is less complex and does not bring an unnecessarily deep hierarchy, which can bring performance boosts on varying sequence lengths. (iii) Moreover, by incorporating the contrastive learning strategy, TDPatcher can better align equivalent suboptimal implementations across different code blocks. The semantic and discriminative features learned from the triplet inputs assist TDPatcher in detecting *TODO-missed methods*. The aforementioned techniques collectively contribute to the remarkable performance of our proposed model.

Answer to RQ-1: How effective is TDPatcher for *TODO-missed methods* detection? We conclude that our approach is highly effective for the task of detecting *TODO-missed methods* in software projects.

4.3 RQ-2. How Effective Is TDPatcher for TODO-missed Methods Patching?

4.3.1 Experimental Setup. In this research question, we want to evaluate the effectiveness of our approach for patching the *TODO-missed methods*. In other words, if a *TODO-missed method* needs to be updated (i.e., adding the TODO comment), whether TDPatcher can patch the TODO comment to the right position. In particular, for each testing triplet sample $\langle \mathbf{m_a}, \mathbf{m_p}, \mathbf{m_n} \rangle$, since the anchor method $\mathbf{m_a}$ and the positive method $\mathbf{m_p}$ contain the same TODO comment, we take the anchor method $\mathbf{m_a}$ as the *TODO-introduced method* and the positive method $\mathbf{m_p}$ as our ground truth (i.e., *TODO-introduced method*). Given the *TODO-introduced method* $\mathbf{m_a}$, we evaluate our approach by checking how often the expected patching position of the *TODO-missed method* $\mathbf{m_p}$ can be accurately found and recommended. To be more specific, we first divide the positive method $\mathbf{m_p}$ into a list of code blocks sequentially. We then pair the anchor code block of $\mathbf{m_a}$ with each divided code block of $\mathbf{m_p}$ and estimate a similarity score by applying our model. The divided code blocks are ranked by their similarity scores for recommendation. If the expected patching position lies within the recommended code block, then we consider TDPatcher patches the *TODO-missed method* $\mathbf{m_p}$ successfully. Notably, we ignore the ±2 lines differences for the task of *TODO-missed*

Table 2. TODO-missed Methods Patching Evaluation

Measure	NCS	CB_B	Ours	Measure	NCS	CB_B	Ours
P@1	50.4%	60.8%	71.0%	DCG@1	50.4%	60.8%	71.0%
P@2	60.0%	71.3%	81.1%	DCG@2	56.4%	67.4%	77.3%
P@3	66.3%	76.5%	86.8%	DCG@3	59.6%	70.0%	80.2%
P@4	70.5%	79.3%	90.3%	DCG@4	61.4%	71.2%	81.7%
P@5	74.0%	82.7%	92.5%	DCG@5	62.8%	72.6%	82.6%

method patching, because we easily determine the exact TODO patching position once the target code block is accurately suggested.

4.3.2 Evaluation Metrics and Baselines. The *TODO-missed method* patching task is a ranking problem, we thus adopted the widely-accepted metric, **P@K** [31] and **DCG@K** [25] to measure the ranking performance of our model. **P@K** is the precision of the expected patching position in top-K recommended locations. Different from **P@K**, **DCG@K** gives a higher reward for ranking the correct element at a higher position. To demonstrate the effectiveness of our TDPatcher for *TODO-missed method* patching task, we compare it with the **CodeBERT$_B$** baseline (denoted as **CB_B** for short in Table 2) and NCS baseline due to their superiority among large pre-trained models and similarity matching–based models in RQ-1, respectively.

4.3.3 Experimental Results. The evaluation results of our approach TDPatcher and baseliens (including NCS and **CodeBERT$_B$**) for RQ-2 are summarized in Table 2. From the table, we can see that (i) NCS achieves the worst performance regarding **P@K** and **DCG@K**. For similarity matching–based approaches, they heavily rely on whether similar code snippets can be found and how similar the code snippets are. Since equivalent suboptimal implementations may have different contexts, it is difficult to identify the corresponding suboptimal locations by simply measuring text similarities. (ii) TDPatcher can patch the TODO comment to the *TODO-missed method* effectively. For example, for a given *TODO-missed method*, TDPatcher can successfully retrieve the right patching position in the first place with a probability of 71%, this score increases up to 92.5% when we enlarge the number of recommendations from 1 to 5, which shows the advantage of our approach for identifying the suboptimal code implementations. (iii) As can be seen, our model TDPatcher is stably and substantially better than the **CodeBERT$_B$** baseline in terms of **P@K** and **DCG@K**. Both the baseline method, i.e., **CodeBERT$_B$**, and our approach can be viewed as variants of embedding algorithm(s), which map code blocks into vectors of a high-dimensional vector space and calculate the similarity scores between vectors. Therefore, the key of *TODO-missed methods* patching relies on how good the generated embeddings are for learning the implicit semantic features. TDPatcher has its advantage over **CodeBERT$_B$** with respect to the following two aspects: First, the embeddings generated by GraphCodeBERT are more suitable than CodeBERT by incorporating the dataflow to capture the inherent structure of code; Moreover, the contrastive learning further adjusts the embeddings for learning equivalent code implementations. The superior performance also verifies the embeddings generated by our approach convey a lot of valuable information.

Answer to RQ-2: How effective is TDPatcher **for *TODO-missed methods* patching?** We conclude that our approach is effective for pinpointing the exact code lines where TODO comments need to be added, which verifies the effectiveness of our approach for the task of *TODO-missed methods* patching.

Table 3. Ablation Evaluation on RQ-1

Measure	CLO	GCBO	TDPatcher
Precision	55.0%	75.0%	83.7%
Recall	65.3%	74.0%	77.3%
F1	59.7%	74.4%	80.4%

Table 4. Ablation Evaluation on RQ-2

Measure	CLO	GCBO	TDPatcher	Measure	CLO	GCBO	TDPatcher
P@1	56.2%	63.1%	71.0%	DCG@1	56.2%	63.1%	71.0%
P@2	65.5%	74.4%	81.1%	DCG@2	62.0%	70.2%	77.3%
P@3	68.8%	80.2%	86.8%	DCG@3	63.7%	73.1%	80.2%
P@4	74.1%	82.3%	90.3%	DCG@4	66.0%	74.0%	81.7%
P@5	76.9%	85.0%	92.5%	DCG@5	67.1%	75.1%	82.6%

4.4 RQ-3. How Effective Is TDPatcher for Using GraphCodeBERT and Contrastive Learning?

4.4.1 Experimental Setup. To better capture the semantic correlations between code blocks and TODO comments, we employ the GraphCodeBERT as encoders and contrastive learning for acquiring suitable vector representations. In this research question, we conduct experiments to verify their effectiveness one by one.

4.4.2 Evaluation Metrics and Baselines. To verify the effectiveness of using GraphCodeBERT and contrastive learning, we compare TDPatcher with the following two baselines on *TODO-missed method* detection (RQ-1) and patching (RQ-2) tasks, respectively, we use the same evaluation metrics for RQ-1 and RQ-2.

- **Contrastive Learning Only (CLO)** removes GraphCodeBERT from our model and only retains contrastive learning. We replace GraphCodeBERT embedding with traditional word embedding techniques (i.e., Glove word vectors). The CLO is then trained with the triplet samples with contrastive learning.
- **GraphCodeBERT only (GCBO)** removes contrastive learning from our model and only keeps the GraphCodeBERT as encoders, the generated vectors of code blocks are used to calculate the similarity score directly without going through the contrastive learning layer. GCBO is similar to **CodeBERT$_B$** baseline except its code embeddings are generated by GraphCodeBERT instead of CodeBERT.

4.4.3 Experimental Results. The experimental results of the ablation analysis on RQ-1 and RQ-2 are summarized in Tables 3 and 4, respectively. It can be seen that (i) No matter which component we removed, it hurts the overall performance of our model, which verifies the usefulness and necessity of using GraphCodeBERT and contrastive learning. (ii) CLO achieves the worst performance. There is a significant drop overall in every evaluation metric after removing the GraphCodeBERT. This signals that the embeddings generated by GraphCodeBERT have a major influence on the overall performance. (iii) The only difference between GCBO and **CodeBERT$_B$** is their code embedding generation model (e.g., GCBO uses GraphCodeBERT while **CodeBERT$_B$** uses CodeBERT). Even though GCBO does not get top results as TDPatcher, it still achieves a considerable performance on RQ-1 and RQ-2 (better than **CodeBERT$_B$** baseline), which further confirms the advantage of GraphCodeBERT embeddings over CodeBERT. (vi) By comparing the performance of GCBO

with our TDPatcher, we can measure the performance improvement achieved by employing the contrastive learning. It is notable that the advantage of our model is more obvious on RQ-2, which shows the benefits of contrastive learning for separating suboptimal code implementations from irrelevant ones.

Answer to RQ-3: How effective is TDPatcher for for using GraphCodeBERT and contrastive learning? We conclude that both GraphCodeBERT and contrastive learning are effective and helpful to enhance the performance of our model.

4.5 RQ-4. How Effective Is TDPatcher under Different Code Block Configurations?

4.5.1 Experimental Setup. In our study, TDPatcher encodes the TODO comment and its associated suboptimal implementations by using code blocks. Each code block is constructed with three segments: namely *TDOO comment*, *TODO centrepiece*, and *TODO context*. Particularly, we choose the code line after the TODO comment as the *TODO centrepiece*, and choose two code lines before and after TODO comment (excluding the *TODO centrepiece*) as the *TODO context*. Our choice of *TODO centrepiece* and *TODO context* is based on empirical observations. There is no assurance that the current configuration of code blocks is definitive and/or optimal. In this research question, we aim to investigate the TDPatcher's performance across various code block configurations.

4.5.2 Evaluation Metrics and Baselines. We estimate TDPatcher's performance by adjusting code block configurations using different *TODO centrepiece* and *TODO context* settings. Specifically, to evaluate the impact of different *TODO centrepiece* choices, we keep the *TODO context* settings consistent with the previous configuration. We then compare the TDPatcher against the following variants on *TODO-missed method* detection (RQ-1) and patching tasks (RQ-2), respectively. The evaluation metrics employed for both RQ-1 and RQ-2 remain the same for a fair comparison.

- **CEN-1:** For this baseline, we use the code line before the TODO comment as the *TODO centrepiece*. The *TODO context* remains consistent with the settings used in TDPatcher. This baseline is denoted as CEN-1.
- **CEN+1:** For this baseline, we use the second code line after the TODO comment as the *TODO centrepiece*. The *TODO context* remains the same settings with TDPatcher. This baseline is denoted as CEN+1.

Similarly, to assess different configurations of *TODO context*, we keep *TODO centrepiece* the same as previous settings (i.e., the code line following the TODO comment) and compare TDPatcher with the following *TODO context* variants.

- **CON_1**: We use one code line before and code one line after the TODO comment (excluding the *TODO centrepiece*) as *TODO context*, this baseline is denoted as CON_1.
- **CON_3**: We use three code lines before and three code lines after the TODO comment (excluding the *TODO centrepiece*) as *TODO context*, this baseline is denoted as CON_3.

4.5.3 Experimental Results. The experimental results of TDPatcher under different code block settings are shown in Tables 5 and 6 for RQ1 and RQ2, respectively. From the table, several points stand out: (i) Regarding the *TODO centerpiece*, we notice that our framework achieves its best performance when the *TODO centerpiece* is set to the code line following the TODO comment. This is reasonable, because putting comments before the code implementations is a common practice for software development. CEN+1 outperforms CEN-1 for both RQ1 and RQ2 also confirms our assumption that the code lines after the TODO comment are more closely related to the suboptimal implementations. (ii) Regarding the *TODO context*, TDPatcher has its advantage as compared to CON_1 and CON_3. CON_1 only includes one code line before and after the TODO comment as

Table 5. Different Code Block Evaluation on RQ-1

Approach	Precision	Recall	F1
CEN-1	78.2%	74.1%	76.1%
CEN+1	78.2%	73.6%	75.7%
CON_1	77.4%	70.6%	73.6%
CON_3	71.4%	74.8%	73.1%
TDPatcher	83.7%	77.3%	80.4%

Table 6. Different Code Block Evaluation on RQ-2

Measure	CEN-1	CEN+1	CON_1	CON_3	TDPatcher
P@1	60.7%	64.5%	65.3%	64.0%	71.0%
P@2	68.8%	76.8%	76.0%	75.1%	81.1%
P@3	74.5%	81.2%	81.1%	80.1%	86.8%
P@4	78.2%	84.3%	84.5%	83.5%	90.3%
P@5	80.7%	87.3%	88.2%	86.9%	92.5%
DCG@1	60.7%	64.5%	65.3%	64.0%	71.0%
DCG@2	65.8%	72.2%	72.1%	71.0%	77.3%
DCG@3	68.7%	74.4%	74.6%	73.5%	80.2%
DCG@4	70.3%	75.8%	76.1%	75.0%	81.7%
DCG@5	71.2%	77.0%	77.5%	76.3%	82.6%

its TODO context, in other words, each code block of CON_1 only contains three code lines. The poor performance of CON_1 shows that it is not sufficient to cover the suboptimal implementations associated with the TODO comment. CON_3 contains three code lines before and after the TODO comment as its *TODO context*. The overall performance decreases when we increase the code block from 5 code lines to 7 code lines. This suggests that larger code block settings bring more noise into the code block in the form of irrelevant code lines, and it thus incurs bigger challenges and difficulties for our task of *TODO-missed Methods* detection and patching.

Answer to RQ-4: How effective is TDPatcher **under different code block configurations?** We conclude that TDPatcher achieves its optimal performance when we set the *TODO centerpiece* as the code line following the TODO comment, and the *TODO context* as two code lines surrounding the *TODO centerpiece*.

4.6 Manual Analysis

To better understand the strengths and limitations of our approach, we manually inspect a number of test results. Example 1 in Figure 6 shows an example where TDPatcher makes correct predictions while other baselines fail. From this example, we can observe that TDPatcher successfully learns to capture the semantic-level features between TODO comments and code implementations, while the textual similarity–based models rely on lexical-level features and are hard to model such correlations. Moreover, contrastive learning further enables our TDPatcher to learn implicit connections among similar code patterns.

We summarize two aspects that cause TDPatcher to experience difficulties. First, one common failed situation is that some suboptimal implementations are too general to be matched, it is very

Example 1 (True Positive— Successful Case)

```
def surname(self):
    # TODO: : Give users the ability to specify this via their profile
    return self.fullname.split(' ')[-1]
```

```
def given_name_initial(self):
    # TODO: : Give users the ability to specify this via their profile
    return self.given_name[0]
```

Example 2 (False Negative — Failed Case)

```
# TODO: complete this method
def get_matvec(self, kshift, imds=None, left= False, **kwargs):
    return None
```

```
# TODO: complete this method
def amplitudes_to_vector (self, r1, r2, kshift, kconserv=None):
    return amplitudes_to_vector_ee (r1, r2, kshift, kconserv)
```

Example 3 (False Positive — Failed Case)

```
def _config_root_Linux():
    key = 'XDG_DATA_HOME' # TODO: use XDG_CONFIG_HOME, ref #99
    root = os.environ.get(key, None) or fallback
    return os.path.join(root, 'python_keyring')
```

```
def _data_root_Linux():
    root = os.environ.get('XDG_DATA_HOME', None) or fallback
    return os.path.join(root, 'python_keyring')
```

Fig. 6. Manual analysis examples.

difficult, if not possible, for TDPatcher to correctly infer such test samples. For example, as shown in Example 2 of Figure 6, the developer added a TODO comment "*TODO: complete this method*" to different unfinished methods at the same time (i.e., `get_matvec` and `amplitudes_to_vector`), which reminds the developer to work on these two methods when time permits. TDPatcher failed to detect `amplitudes_to_vector` missing the above TODO comment, this is because it is not easy to claim this method is completed or not under the current code context.

Another bad situation is that the suboptimal implementations do not provide sufficient information for *TODO-missed methods* identification. Example 3 in Figure 6 shows such a test sample, the developers added a TODO comment, i.e., "*TODO: use XDG_CONFIG_HOME*" to `_config_root_Linux` method to indicate the temporary implementation for the `key` variable. However, this code implementation is suboptimal for the method `_config_root_Linux` while complete for the method `_data_root_Linux`. Considering our approach lacks the contextual information regarding method names, TDPatcher thus mispatched this TODO comment to `_data_root_Linux` method. Addressing these challenges will remarkably boost the learning performance of our model, we will focus on this research direction in the future.

Why does TDPatcher **succeed/fail?** TDPatcher successfully learns to capture the semantical features between TODO comments and their suboptimal implementations, however, it fails to handle if the suboptimal implementations are too general or lacking sufficient information, we will try to address these limitations in our future work.

Ex.1 True Sample: explosion/spaCy

```
76      # TODO: Not sure what's wrong here. Possible bug?
77      @pytest.mark.xfail
78      def test_matcher_match_zero(self):
...
101     # TODO: Not sure what's wrong here. Possible bug?
102     @pytest.mark.xfail
103     def test_matcher_match_one_plus(self): …
```

```
66     @pytest.mark.xfail
67     def test_matcher_phrase_match(en_vocab): …
```

Ex.2 Unsure Sample: cms-dev/cms

```
264     def evaluation_finished(self, success, submission_id):  …
279         self.queue.lock()
280         # TODO: - should check the original priority of the job
281         self.queue.push((EvaluationServer.JOB_TYPE_EVALUATION, submission),
282                         EvaluationServer.JOB_PRIORITY_LOW)
283         self.queue.unlock()
```

```
237     def compilation_finished(self, success, submission_id):  …
247         self.queue.lock()
248         self.queue.push((EvaluationServer.JOB_TYPE_EVALUATION, submission),
249                         priority)
250         self.queue.unlock()
```

Ex.3 False Sample: sunpy/sunpy

```
65      def esp_test_ts():
66          # ToDo: return sunpy.timeseries.TimeSeries(os.path.join(testpath, filename), source='ESP')
67          return sunpy.timeseries.TimeSeries(esp_filepath, source='ESP')
```

```
88      def lyra_test_ts():
89          # ToDo: return sunpy.timeseries.TimeSeries(os.path.join(testpath, filename), source='LYRA')
90          return sunpy.timeseries.TimeSeries(lyra_filepath, source='LYRA')
```

Fig. 7. In-the-wild evaluation examples.

5 In-the-wild Evaluation

Besides the automatic evaluation, we further conduct an in-the-wild evaluation to evaluate the effectiveness of TDPatcher in real-world software projects. To do so, we randomly select 50 Python repositories from our testing dataset, then for each GitHub repository, we checkout all the TODO-introducing commits and extract their associated *TODO-introduced methods*. Note that for building our dataset, we only use *paired TODO-introduced methods* and excluded the *unpaired* ones. For this in-the-wild evaluation, both paired and unpaired *TODO-introduced methods* are utilized. We then apply TDPatcher to pinpoint if any methods' code blocks regarding this snapshot missed the introduced TODO comment. As a result, TDPatcher reports 41 *TODO-missed methods* in total, each *TODO-missed method* and its original *TODO-introduced method* pair are provided to three experts (all of which have Python programming experience for more than 5 years) for evaluation independently. Each expert manually labels the reported method as *True* or *False* or *Unsure* independently. The final results are determined by the majority rule of voting (if three different scores are obtained, the sample will be regarded as Unsure). Finally, after the manual checking, 26 methods are labeled as True (i.e., *TODO-missed methods*), 10 methods are labeled as False, and 5 methods are labeled as Unsure. We have published the in-the-wild evaluation results in our replication package for verification [1].

We demonstrated three *TODO-missed methods* detected by our TDPatcher in Figure 7. The Example 1 shows a True sample. In particular, after the developer ran the test case, several test methods failed (e.g., `test_matcher_match_zero` and `test_matcher_match_one_plus`). To achieve short-term benefits (e.g., higher productivity or shorter release time), the developer marked these methods with the `xfail` marker (e.g., line 77/78, `@pytest.mark.xfail`). As a result, the details of

these tests will not be printed even if they fail. The developers added TODO comments here (e.g., Line 76/101) to remind themselves of these markers. Later, when they figured out the tests failed reasons, these markers can be removed. However, the same situation also applied to the method `test_matcher_phrase_match`, but the corresponding TODO comment was forgotten to be added, which may cause the test method to be neglected.

The Example 2 demonstrates an Unsure sample. The developer added a TODO comment (*"TODO: - should check the original priority of the job"*) to `evluation_finished` method. Our approach identified another `compilation_finished` as a *TODO-missed method* because of their similar local code implementations. Due to unfamiliarity with the code, it is challenging, even for human experts, to annotate such samples. For such cases, we still need developers to double-check the results. However, we argue that it is still beneficial to prompt the developers to be aware of such potential suboptimal places in time after they introduce any new TODOs.

Example 3 demonstrates a common False sample in our in-the-wild evaluation. Given the *TODO-introduced method* `esp_test_ts` with its added TODO comment (i.e., Line 66), TDPatcher wrongly predicted this TODO comment should also be added to the `lyra_test_ts` method. The failed reason behind this sample is that the TODO comments within these two methods are slightly "different", while our approach can only handle methods missing the exactly **same** TODO comment. This sample also points out an interesting improvement for our approach, i.e., after patching the TODO comment to *TODO-missed methods*, it is necessary to further *update* the TODO comment according to a different method context, which could be handled by incorporating the comment updating tools [29, 32, 43]. We will explore this research direction in our future work.

How effective is TDPatcher for detecting and patching *TODO-missed methods* in the wild? TDPatcher successfully detects 26 *TODO-missed methods* from 50 Github repositories, which shows the effectiveness for detecting and patching *TODO-missed methods* in real-world Github repositories.

6 Threats to Validity

Several threats to validity are related to our research:

Internal Validity. Threats to internal validity relate to the potential errors in our code implementation and study settings. Regarding the data preparation process, to ensure the quality of our constructed dataset, we only consider code snippets with the same TODO comments as equivalent suboptimal implementations, we removed TODO comments if they do not contain exactly the same words. In practice, similar TODO comments could also be associated with equivalent suboptimal implementations (as shown in Figure 7 Example 3). These instances are missed by our data collection process. Detecting and adding such missed cases can enlarge our dataset and boost our model performance, we will focus on this direction in the future. We choose Python 2.7 and 3.7 standard library `ast` for parsing Python2 and Python3 projects, there may exist methods with new syntax that our parser cannot handle. Only 1.5% (1,451 of 97,413) methods failed to parse due to syntax errors, the potential threats of syntax errors are limited. Since we collected our dataset from top-10K Python projects that may contain noise, we further performed a manual validation, ensuring the quality of our constructed dataset. Regarding the evaluation settings, we evaluate our approach by extracting methods only from *TODO-introduced* files, which is simpler than real-world situations. We consider this simplification as a tradeoff to estimate our model's effectiveness and simulate real-world working environments, since more than 70% *TODO-introduced methods* are added within the same file. Expanding our approach to other files could introduce new challenges (i.e., data imbalance) and will be our future research direction. To reduce errors in automatic

evaluation, we have double-checked the source code, we have carefully tuned the parameters of the baseline approaches and used the highest-performing settings for each approach. Considering there may still exist errors that we did not note or neglect, we have published our source code to facilitate other researchers to replicate and extend our work.

External Validity. Threats to external validity are concerned with the generalizability of our study. One of the external validities is the dataset, our dataset is constructed from Python projects in GitHub. This is because Python is one of the most popular programming languages widely used by developers in GitHub. Considering that our approach is language-independent, we believe that our approach can be easily adapted to other programming languages. Another external validity is that our work focused on TODO comment, which is a special case of **Self-Admitted Technical Debt (SATD)** [46]. Our preliminary study focuses on TODOs as opposed to other SATD (e.g., HACK, FIXME, and NOTE). This is because TODO comments construct the major proportion of SATD. We will try to extend our work to other programming languages as well as other types of SATD to benefit more developers in the future.

Model Validity. Threats to model validity relate to model structure that could affect the learning performance of our approach. In this study, we choose the state-of-the-art code embedding pre-trained model, GraphCodeBERT, for our approach. GraphCodeBERT was pretrained on the CodeSearchNet dataset, which includes 2.3M functions of six programming languages paired with natural language documents. Our study focuses on methods related to TODOs, which means these methods' implementations are temporarily suboptimal. We believe that the potential threats of data leakage associated with employing GraphCodeBERT for our specific tasks are minimal. Recent research has proposed new models, such as codet5 [59], copilot [5], and alphacode [27] for different advanced programming tasks (e.g., code generation, competitive programming, etc). Our approach does not shed light on the effectiveness of employing other advanced pre-trained models with respect to new structures and new features. We will explore other advanced models in future work.

7 Related Work

7.1 TODO Comments in Software Engineering

TODO comments are extensively used by developers as reminders to describe the temporary code solutions that should be revisited in future. Despite the fact that TODO comments are widely adopted, the research works focus on TODO comments are still very limited. Previous studies have investigated the TODO comments usage in software development and maintenance [17, 42, 43, 52, 53].

Storey et al. [53] performed an empirical study among developers to investigate how TODO comments are used and managed in software engineering. They found that the use of task annotations varies from individuals to teams, and if incorrectly managed, then they could negatively impact the maintenance of the system. Sridhara et al. [52] developed a technique to check the status of the TODO comments, their approach automatically checks if a TODO comment is up to date by using the information retrieval, linguistics and semantics methods. Nie et al. [42] investigated several techniques based on natural language processing and program analysis techniques that have the potential to substantially simplify software maintenance and increase software reliability. Following that, they proposed a framework, TrigIt, to encode trigger action comments as executable statements [43]. Experimental results show that TrigIt has the potential to enforce more discipline in writing and maintaining TODO comments in large code repositories. Gao et al. [17] proposed a deep-learning-based approach TDCleaner to detect and remove obsolete TODO comments just-in-time by mining the commit histories of the software repositories.

Different from the previous research, in this work we propose a novel task of *TODO-missed methods* detection and patching. We build the first large dataset for this task and explore the possibility of automatically adding TODO comments to the *TODO-missed* methods.

7.2 Self-admitted Technical Debt in Software Engineering

TODO comment is a common type of SATD. SATD was first proposed by Potdar and Shihab [46] in 2014, which refers to the technical debt (i.e., code that is incomplete, defective, temporary and/or suboptimal) consciously introduced by developers and documented in the form of comments they self-admit it. In recent years, various studies have investigated SATD from different aspects [50], i.e., SATD detection, SATD comprehension and SATD repayment [3, 7–9, 11, 20, 23, 24, 28, 30, 36–38, 40, 44–46, 60, 62, 63, 66–69].

Regarding SATD detection, Potdar et al. [7] extracted 101,762 code comments from m 4 large open source systems, and manually identified 62 patterns that indicate SATD. Maldonado et al. [8] presented an **Natural Language Processing– (NLP)** based method to automatically identify design and requirement self-admitted technical debt. Huang et al. [23] used text-mining-based methods to predict whether a comment contains SATD or not. Yan et al. [64] first proposed the change-level SATD determination model by extracting 25 change features, their model can determine whether or not a software change introduces TD when it is submitted. Regarding SATD comprehension, different studies have been conducted to understand the SATD in software development life cycles. For example, Maldonado et al. [36] manually analyzed 33,093 comments and classified them into five types (i.e., design debt, defect debt, documentation debt, requirement debt and test debt). Bavota et al. [4] studied the introduced, removed, and unaddressed SATDs in projects' change history. Wehaibi et al. [60] empirically studied how technical debt relates to software quality from five large open source software projects. Xiao et al. [63] conducted a qualitative analysis of 500 SATD comments in the Maven build system to better understand the characteristics of SATD in build systems. Regarding the SATD repayment, prior works study the removal of SATD from software systems, i.e., repaying the SATDs. Zampetti et al. [69] conducted an in-depth study on the removal of SATD. They found that 25% to 60% SATD were removed due to full class or method removal, and 33% to 63% SATD were removed as the partial change in their corresponding method. Most recently, Rungroj et al. [35] introduced the concept of "on-hold" SATD and proposed a tool [34] to identify and remove the "on-hold" SATD automatically. Mastropaolo et al. [37] empirically investigated the extent to which technical debt can be automatically paid back by neural-based generative models. Alhefdhi et al. [3] proposed a deep learning model, named DLRepay, for automated SATD repayment.

Different from the aforementioned research, our work focuses on TODO-related methods. This is because the TODO comments are regarded as the most common type of SATD, which are widely used by developers across different software projects. Moreover, TDPatcher can be easily transferred to other types of STAD (e.g., FIXME, HACK), we plan to adapt our model to other SATD in our future work.

7.3 Code Embeddings in Software Engineering

Embedding is a technique from NLP that learns distributed vector representation for different entities (e.g., words and sentences). One of the typical embedding techniques is word2vec [39], which encodes each word as a fixed-size vector, where similar words are close to each other in vector space. Recently, an attractive research direction in software engineering is to learn vector representations of source code [5, 12, 14, 15, 19, 59] via large pre-trained models.

Feng et al. [12] proposed CodeBERT, which learns the vector representation from **programming languages (PL)** and **natural language (NL)** for downstream code-related tasks, e.g., code

search and code documentation generation tasks. Different from the existing pre-trained models regard a code snippet as a sequence of tokens, Guo et al. [19] presented GraphCodeBERT, which uses dataflow in the pre-training stage and learns the inherent structure of code. The GraphCodeBERT has shown its effectiveness on four code-related tasks, including code search, clone detection, code translation, and code refinement. Wang et al. [59] proposed CodeT5, which employs a unified framework to seamlessly support both code understanding and generation tasks (i.e., PL-NL, NL-PL, and PL-PL).

In this research, we first adopt the GraphCodeBERT for the task of TODO-missed methods detection and patching, which provides contextualized embeddings for TODO comments and their associated suboptimal implementations.

8 Conclusion and Future Work

This research first propose the task of automatically detecting and patching *TODO-missed methods* in software projects. To solve this task, we collect *TODO-introduced methods* from the top-10,000 GitHub repositories. To the best of our knowledge, this is the first large dataset for *TODO-missed methods* detection. We propose an approach named TDPatcher, which leverage a neural network model to learn the semantic features and correlations between TODO comments and their suboptimal implementations. Extensive experiments on the real-world GitHub repositories have demonstrated effectiveness and promising performance of our TDPatcher. In the future, we plan to investigate the effectiveness of TDPatcher with respect to other programming languages. We also plan to adapt TDPatcher to other types of task comments, such as FIXME, HACK.

Acknowledgments

The authors thank the reviewers for their insightful and constructive feedback.

References

[1] [n. d.]. TDPatcher: Our Replication Package. Retrieved from https://github.com/TDPatcher/TDPatcher

[2] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 2655–2668.

[3] Abdulaziz Alhefdhi, Hoa Khanh Dam, and Aditya Ghose. 2023. Towards automating self-admitted technical debt repayment. *Inf. Softw. Technol.* 167, C (2023), 107376.

[4] Gabriele Bavota and Barbara Russo. 2016. A large-scale empirical study on self-admitted technical debt. In *Proceedings of the 13th International Conference on Mining Software Repositories*. 315–326.

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. arXiv:2107.03374. Retrieved from https://arxiv.org/abs/2107.03374

[6] Tao Chen, Haizhou Shi, Siliang Tang, Zhigang Chen, Fei Wu, and Yueting Zhuang. 2021. CIL: Contrastive instance learning framework for distantly supervised relation extraction. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. 6191–6200.

[7] Michael L. Collard, Michael J. Decker, and Jonathan I. Maletic. 2011. Lightweight transformation and fact extraction with the srcML toolkit. In *Proceedings of the IEEE 11th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 173–184.

[8] Everton da Silva Maldonado, Emad Shihab, and Nikolaos Tsantalis. 2017. Using natural language processing to automatically detect self-admitted technical debt. *IEEE Trans. Softw. Eng.* 43, 11 (2017), 1044–1062.

[9] Mário André de Freitas Farias, Manoel Gomes de Mendonça Neto, Marcos Kalinowski, and Rodrigo Oliveira Spínola. 2020. Identifying self-admitted technical debt through code comment analysis with a contextualized vocabulary. *Inf. Softw. Technol.* 121 (2020), 106270.

[10] Xingping Dong and Jianbing Shen. 2018. Triplet loss in siamese network for object tracking. In *Proceedings of the European Conference on Computer Vision (ECCV'18)*. 459–474.

[11] Neil A Ernst, Stephany Bellomo, Ipek Ozkaya, Robert L Nord, and Ian Gorton. 2015. Measure it? Manage it? Ignore it? Software practitioners and technical debt. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. 50–60.

[12] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP'20*. 1536–1547.

[13] Tianyu Gao, Xingcheng Yao, and Danqi Chen. 2021. SimCSE: Simple contrastive learning of sentence embeddings. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. 6894–6910.

[14] Zhipeng Gao, Vinoj Jayasundara, Lingxiao Jiang, Xin Xia, David Lo, and John Grundy. 2019. Smartembed: A tool for clone and bug detection in smart contracts through structural code embedding. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'19)*. IEEE, 394–397.

[15] Zhipeng Gao, Xin Xia, John Grundy, David Lo, and Yuan-Fang Li. 2020. Generating question titles for stack overflow from mined code snippets. *ACM Trans. Softw. Eng. Methodol.* 29, 4 (2020), 1–37.

[16] Zhipeng Gao, Xin Xia, David Lo, John Grundy, Xindong Zhang, and Zhenchang Xing. 2023. I know what you are searching for: Code snippet recommendation from Stack Overflow posts. *ACM Trans. Softw. Eng. Methodol.* 32, 3 (2023), 1–42.

[17] Zhipeng Gao, Xin Xia, David Lo, John Grundy, and Thomas Zimmermann. 2021. Automating the removal of obsolete TODO comments. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 218–229.

[18] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of the IEEE/ACM 40th International Conference on Software Engineering (ICSE'18)*. IEEE, 933–944.

[19] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, LIU Shujie, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. GraphCodeBERT: Pre-training code representations with data flow. In *Proceedings of the International Conference on Learning Representations*.

[20] Zhaoqiang Guo, Shiran Liu, Jinping Liu, Yanhui Li, Lin Chen, Hongmin Lu, and Yuming Zhou. 2021. How far have we progressed in identifying self-admitted technical debts? A comprehensive empirical study. *ACM Trans. Softw. Eng. Methodol.* 30, 4 (2021), 1–56.

[21] Raia Hadsell, Sumit Chopra, and Yann LeCun. 2006. Dimensionality reduction by learning an invariant mapping. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, Vol. 2. IEEE, 1735–1742.

[22] Elad Hoffer and Nir Ailon. 2015. Deep metric learning using triplet network. In *Proceedings of the 3rd International Workshop on Similarity-Based Pattern Recognition (SIMBAD'15)*. Springer, 84–92.

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

[24] Martina Iammarino, Fiorella Zampetti, Lerina Aversano, and Massimiliano Di Penta. 2021. An empirical study on the co-occurrence between refactoring actions and self-admitted technical debt removal. *J. Syst. Softw.* 178 (2021), 110976.

[25] Kalervo Järvelin and Jaana Kekäläinen. 2002. Cumulated gain-based evaluation of IR techniques. *ACM Trans. Inf. Syst.* 20, 4 (2002), 422–446.

[26] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *Proceedings of the 37th International Conference on Machine Learning*. 5110–5121.

[27] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.

[28] Zengyang Li, Paris Avgeriou, and Peng Liang. 2015. A systematic mapping study on technical debt and its management. *J. Syst. Softw.* 101 (2015), 193–220.

[29] Bo Lin, Shangwen Wang, Kui Liu, Xiaoguang Mao, and Tegawendé F. Bissyandé. 2021. Automated comment update: How far are we? In *Proceedings of the IEEE/ACM 29th International Conference on Program Comprehension (ICPC'21)*. IEEE, 36–46.

[30] Jiakun Liu, Qiao Huang, Xin Xia, Emad Shihab, David Lo, and Shanping Li. 2020. Is using deep learning frameworks free? Characterizing technical debt in deep learning frameworks. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Society*. 1–10.

[31] Tie-Yan Liu, et al. 2009. Learning to rank for information retrieval. *Found. Trends Inf. Retriev.* 3, 3 (2009), 225–331.

[32] Zhongxin Liu, Xin Xia, Meng Yan, and Shanping Li. 2020. Automating just-in-time comment updating. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 585–597.

[33] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. CodeXGLUE: A machine learning benchmark dataset for code understanding and

generation. In *Proceedings of the 35th Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

[34] Rungroj Maipradit, Bin Lin, Csaba Nagy, Gabriele Bavota, Michele Lanza, Hideaki Hata, and Kenichi Matsumoto. 2020. Automated identification of on-hold self-admitted technical debt. In *Proceedings of the IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM'20)*. IEEE, 54–64.

[35] Rungroj Maipradit, Christoph Treude, Hideaki Hata, and Kenichi Matsumoto. 2020. Wait for it: Identifying "On-Hold" self-admitted technical debt. *Emp. Softw. Eng.* 25, 5 (2020), 3770–3798.

[36] Everton da S. Maldonado and Emad Shihab. 2015. Detecting and quantifying different types of self-admitted technical debt. In *Proceedings of the IEEE 7Th International Workshop on Managing Technical Debt (MTD'15)*. IEEE, 9–15.

[37] Antonio Mastropaolo, Massimiliano Di Penta, and Gabriele Bavota. 2023. Towards automatically addressing self-admitted technical debt: How far are we? In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE'23)*. IEEE, 585–597.

[38] Ana Melo, Roberta Fagundes, Valentina Lenarduzzi, and Wylliams Barbosa Santos. 2022. Identification and measurement of requirements technical debt in software development: A systematic literature review. *J. Sys. Softw.* 194 (2022), 111483.

[39] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. arXiv:1301.3781. Retrieved from https://arxiv.org/abs/1301.3781

[40] Biruk Asmare Muse, Csaba Nagy, Anthony Cleve, Foutse Khomh, and Giuliano Antoniol. 2022. FIXME: Synchronize with database! An empirical study of data access self-admitted technical debt. *Emp. Softw. Eng.* 27, 6 (2022), 130.

[41] Jaechang Nam, Sinno Jialin Pan, and Sunghun Kim. 2013. Transfer defect learning. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*. IEEE, 382–391.

[42] Pengyu Nie, Junyi Jessy Li, Sarfraz Khurshid, Raymond Mooney, and Milos Gligoric. 2018. Natural language processing and program analysis for supporting todo comments as software evolves. In *Workshops at the 32nd AAAI Conference on Artificial Intelligence*.

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

[44] Sheena Panthaplackel, Pengyu Nie, Milos Gligoric, Junyi Jessy Li, and Raymond Mooney. 2020. Learning to update natural language comments based on code changes. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 1853–1868.

[45] Saranphon Phaithoon, Supakarn Wongnil, Patiphol Pussawong, Morakot Choetkiertikul, Chaiyong Ragkhitwetsagul, Thanwadee Sunetnanta, Rungroj Maipradit, Hideaki Hata, and Kenichi Matsumoto. 2021. FixMe: A GitHub bot for detecting and monitoring on-hold self-admitted technical debt. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE'21)*. IEEE, 1257–1261.

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

[47] Fangcheng Qiu, Zhipeng Gao, Xin Xia, David Lo, John Grundy, and Xinyu Wang. 2021. Deep just-in-time defect localization. *IEEE Trans. Softw. Eng.* 48, 12 (2021), 5068–5086.

[48] Chanchal K. Roy and James R. Cordy. 2008. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proceedings of the 16th IEEE International Conference on Program Comprehension*. IEEE, 172–181.

[49] Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. 2018. Retrieval on source code: A neural code search. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 31–41.

[50] Giancarlo Sierra, Emad Shihab, and Yasutaka Kamei. 2019. A survey of self-admitted technical debt. *J. Syst. Softw.* 152 (2019), 70–82.

[51] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. 2018. PyDriller: Python framework for mining software repositories. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'18)*. ACM Press, New York, NY, 908–911. https://doi.org/10.1145/3236024.3264598

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

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

[54] Yanqi Su, Zheming Han, Zhipeng Gao, Zhenchang Xing, Qinghua Lu, and Xiwei Xu. 2023. Still confusing for bug-component triaging? Deep feature learning and ensemble setting to rescue. In *Proceedings of the IEEE/ACM 31st International Conference on Program Comprehension (ICPC'23)*. IEEE, 316–327.

[55] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. 2016. Automated parameter optimization of classification techniques for defect prediction models. In *Proceedings of the 38th International Conference on Software Engineering*. 321–332.

[56] Chenning Tao, Qi Zhan, Xing Hu, and Xin Xia. 2022. C4: Contrastive cross-language code clone detection. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. 413–424.

[57] Yao Wan, Wei Zhao, Hongyu Zhang, Yulei Sui, Guandong Xu, and Hai Jin. 2022. What do they capture? A structural analysis of pre-trained language models for source code. In *Proceedings of the 44th International Conference on Software Engineering*. 2377–2388.

[58] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically learning semantic features for defect prediction. In *Proceedings of the IEEE/ACM 38th International Conference on Software Engineering (ICSE'16)*. IEEE, 297–308.

[59] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 8696–8708.

[60] Sultan Wehaibi, Emad Shihab, and Latifa Guerrouj. 2016. Examining the impact of self-admitted technical debt on software quality. In *Proceedings of the IEEE 23Rd International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*, Vol. 1. IEEE, 179–188.

[61] Moshi Wei, Nima Shiri Harzevili, Yuchao Huang, Junjie Wang, and Song Wang. 2022. CLEAR: Contrastive learning for API recommendation. In *Proceedings of the IEEE/ACM 44th International Conference on Software Engineering (ICSE'22)*. IEEE, 376–387.

[62] Laerte Xavier, Fabio Ferreira, Rodrigo Brito, and Marco Tulio Valente. 2020. Beyond the code: Mining self-admitted technical debt in issue tracker systems. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 137–146.

[63] Tao Xiao, Dong Wang, Shane McIntosh, Hideaki Hata, Raula Gaikovina Kula, Takashi Ishio, and Kenichi Matsumoto. 2021. Characterizing and mitigating self-admitted technical debt in build systems. *IEEE Trans. Softw. Eng.* 48, 10 (2021), 4214–4228.

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

[65] Yanming Yang, Xing Hu, Zhipeng Gao, Jinfu Chen, Chao Ni, Xin Xia, and David Lo. 2024. Federated learning for software engineering: A case study of code clone detection and defect prediction. *IEEE Trans. Softw. Eng.* (2024).

[66] Zhe Yu, Fahmid Morshed Fahid, Huy Tu, and Tim Menzies. 2020. Identifying self-admitted technical debts with jitterbug: A two-step approach. *IEEE Trans. Softw. Eng.* 48, 5 (2020), 1676–1691.

[67] Fiorella Zampetti, Gianmarco Fucci, Alexander Serebrenik, and Massimiliano Di Penta. 2021. Self-admitted technical debt practices: A comparison between industry and open-source. *Emp. Softw. Eng.* 26 (2021), 1–32.

[68] Fiorella Zampetti, Cedric Noiseux, Giuliano Antoniol, Foutse Khomh, and Massimiliano Di Penta. 2017. Recommending when design technical debt should be self-admitted. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'17)*. IEEE, 216–226.

[69] Fiorella Zampetti, Alexander Serebrenik, and Massimiliano Di Penta. 2018. Was self-admitted technical debt removal a real removal? An in-depth perspective. In *Proceedings of the IEEE/ACM 15th International Conference on Mining Software Repositories (MSR'18)*. IEEE, 526–536.

[70] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering (ICSE'19)*. IEEE, 783–794.

[71] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. 2009. Cross-project defect prediction: A large scale experiment on data vs. domain vs. process. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 91–100.

Received 30 December 2022; revised 8 January 2024; accepted 15 February 2024