COLARE: Commit Classification via Fine-grained Context-aware Representation of Code Changes

Qunhong Zeng
School of Computer Science & Technology
Beijing Institute of Technology
Beijing, China
qunhongzeng@bit.edu.cn

Yuxia Zhang‡
School of Computer Science & Technology
Beijing Institute of Technology
Beijing, China
yuxiazh@bit.edu.cn

Zeyu Sun
Institute of Software
Chinese Academy of Sciences
Beijing, China
szy_@pku.edu.cn

Yujie Guo
School of Computer Science & Technology
Beijing Institute of Technology
Beijing, China
guoyujie@bit.edu.cn

Hui Liu
School of Computer Science & Technology
Beijing Institute of Technology
Beijing, China
liuhui08@bit.edu.cn

***Abstract*—Commit classification for maintenance activities is of critical importance for both industry and academia. State-of-the-art approaches either treat code changes as plain text or rely on manually identified features. Directly applying the most advanced model of code change representation into commit classification faces two limitations: (1) coarse-grained diff comparison neglects the distance of modified code lines; (2) missing key context information of hunk modification and file categories. This study proposes a novel classification model, COLARE, which compares code changes at the hunk level, takes fine-grained features based on categories of changed files, and aggregates with the representation of commit messages. The evaluation results show that our model outperforms state-of-the-art techniques by 7.24% and 7.35% in accuracy and macro F1 score, respectively. We also manually labeled a multi-language dataset and evaluated our approach, The results further confirm that our approach achieves the best performance over three baselines, including ChatGPT (3.5). The evaluation of the ablation study demonstrates the effectiveness of the major components in our technique.**

***Index Terms*—Maintenance Activities, Commit Classification, Fine-grained Code Change Representation**

I. Introduction

To keep the hearts beating, software systems rely on continuous changes to reflect new environments, business reorientation, or modernization [12]. Version control systems (VCSs), such as Git, serve as one critical infrastructure supporting effective team collaboration and maintaining a meticulous, traceable history of changes [36]. Developers perform different maintenance activities on VCSs, such as fixing bugs or implementing new features. The resulting code changes are *commit* to version control systems. As evidence of the significant scale of software development activities, the year 2022 witnessed an astonishing number of approximately 800 million commits made by developers worldwide.[1]

As the fundamental unit for representing software changes, commits provide an efficient means of tracking changes made to code repositories [32]. Researchers have categorized commits into various maintenance activity types, with the most commonly used typology comprising three categories: corrective, perfective, and adaptive [40, 32]. Categorizing commits into maintenance activities is of critical importance. In the realm of industry, classified commits help support cost-effective management and evolution of software projects [13, 40, 28]. For instance, software projects typically construct their release notes by mainly checking and categorizing change types [46], such as what new features were implemented and which bugs were fixed. Automatically classifying commits into maintenance activities can significantly improve the efficiency of preparing release notes [23]. Within the sphere of research, considering commit categories allows for measuring developer capabilities in a finer grained, instead of directly calculating commit numbers [3]. Additionally, it facilitates characterizing companies' performance in the domain of open source software [48, 49].

A commit typically comprises the code changes (known as *diff*) and an accompanying descriptive text (known as *message*) that provides context for the changes made. Based on these components, various studies have proposed approaches for automatically classifying commits into maintenance activities. The first research stream of classifying commits involves using keywords (e.g., "fix") in commit messages [32, 2, 18, 28] and/or extracting features from the code diff (e.g., the number of added code lines) [28, 18]. As natural language processing (NLP) techniques advanced, the second stream of work started applying these techniques to enhance commit classification performance. For instance, Ghadhab et al. [13] employed BERT [8] to extract features from commit messages, while Lee et al. [27] utilized CodeBERT [10] to directly encode code diffs.

Leveraging pre-trained language models may present commit messages well but has limitations in extracting features from code changes, because: 1) current approaches treating diff as plain text fail to learn the hierarchical structure information

‡ Corresponding Author
[1] https://bitbucket.org/swsc/overview/src/master/

2640-7574/24/$31.00 ©2024 IEEE
DOI 10.1109/SANER60148.2024.00082

from code diff; and 2) they do not explicitly seize the code changes between the old and new versions, which determine the categories of maintenance activities by nature. Thus, it is tempting to leverage state-of-the-art code change presentation technique, i.e., CC2Vec [19], to address the two limitations. CC2Vec models the hierarchical structure of a code change by the attention mechanism and uses multiple comparison functions to identify the differences between the old and new versions explicitly. However, CC2Vec compares code changes on a coarse-grained level, i.e., directly comparing two embedding vectors representing removed and added lines in a modified file, without paying attention to the distances of the changed lines. Git uses hunk to represent a cohesive set of modifications [11]. Modified lines in different hunks may have limited relation, and the overall comparison of added and deleted lines can introduce noise (see Section II-A for more details). Moreover, CC2Vec misses key context information of code changes, i.e., the type of modified files and the contextual information of changed lines in each hunk, which may be unnecessary to code representation but of critical importance to commit classification (see Section II-B for more details).

In this study, we propose a novel commit classification model, COLARE, which extracts features from code changes at the hunk level while considering the context of code changes. Moreover, we enhance our approach with fine-grained features based on the categories of modified files and then aggregate them with the representation of commit messages. Our model is the first approach that boosts the performance of commit classification by combining the strengths of commit messages, code diff, and file category features.

We conduct extensive evaluations to explore the effectiveness of COLARE. We first compare COLARE with the approach proposed by Ghadhab et al. [13] on their dataset, because this baseline approach is specific to Java projects and achieves state-of-the-art performance. The evaluation results show that COLARE outperforms the state-of-the-art by 7.24% in accuracy and 7.35% in macro F1 score. Since our approach is not limited to Java, we further evaluate its performance in a multiple programming language dataset, which contains $1,581$ commits and is labeled by this paper's first and fourth authors. The results demonstrate the superiority of our model, with an accuracy of 83.37% and a macro F1 score of 78.89%. We also conducted comparisons with three baselines, and the improvements achieved by COLARE range from 4.69% to 33.02%, 4.9% to 28.83% in accuracy and macro F1 score, respectively. In the last, we conduct an ablation study to delve deeper into the necessity of adapting CC2Vec and the effectiveness of the three key components of our approach, i.e., commit messages, code diff, and file category features. The results unequivocally confirm the positive contributions of all components to the overall effectiveness of COLARE.

The main contributions of this work are summarized as follows:

1) A hunk-level code change representation for commit classification.
2) A labeled dataset of $1,581$ commits that were randomly sampled from seven prominent programming language repositories.
3) A hybrid commit classification approach that outperforms the state-of-the-art. The approach is not specific to Java programming language and first combines three sources of key information to classify commits.

II. Motivation

State-of-the-art approaches for commit classification fail to leverage the hierarchical structure information and do not explicitly capture the changes between old and new versions. Although the code change representation produced by CC2Vec [19] has the potential to address these issues, applying CC2Vec directly to commit classification has two limitations, i.e., coarse-grained comparison and key context information missing.

```
django/utils/cache.py
@@ -13,14 +13,15 @@

different header content for headers named in "Vary" need to get different
cache keys to prevent delivery of wrong content.
-   A example: i18n middleware would need to distinguish caches by the
+  An example: i18n middleware would need to distinguish caches by the

@@ -95,8 +96,7 @@ def patch_response_headers(response, cache_timeout=None):
def add_never_cache_headers(response):
    """
-    Add headers to a response to indicate that
-    a page should never be cached.
+    Adds headers to a response to indicate that a page should never be cached.
    """
@@ -139,7 +140,8 @@ def get_cache_key(request, key_prefix=None):
    """
    if key_prefix is None:
        key_prefix = settings.CACHE_MIDDLEWARE_KEY_PREFIX
-    cache_key = 'views.decorators.cache.cache_header.%s.%s' % (key_prefix,
iri_to_uri(request.path))
+    cache_key = 'views.decorators.cache.cache_header.%s.%s' % (
+                    key_prefix, iri_to_uri(request.path))
    headerlist = cache.get(cache_key, None)
    if headerlist is not None:
        return _generate_cache_key(request, headerlist, key_prefix)
```

Fig. 1: **Motivating example: A commit from Django**

A. Limitation 1: Coarse-grained Comparison

The first challenge lies in CC2Vec's tactic of comparing the removed and added code lines at the file granularity. This approach cannot exploit the inherent comparative information available at the hunk level, making it arduous to capture fine-grained differences in the code diff. We illustrate this point with an example shown in Figure 1, which represents the code diff within a single file in a real-world commit[2] from Django, a popular web framework project [3]. Git uses hunk to represent a cohesive set of modifications that can be applied or reverted as a single unit [11]. In Figure 1, we can see that there are three hunks: the first hunk appears at line 13, the second hunk is positioned at line 95, and the third hunk begins at line 139. Figure 2 shows how CC2Vec extracts features from the code diff shown in Figure 1. Specifically, CC2Vec extracts all embeddings of deleted and added code lines from each hunk. Subsequently, it employs an attention mechanism to

[2]https://github.com/django/django/commit/d453eda38852b1fb4f10a03a467abae8ca4ae927

[3]https://www.djangoproject.com/

reduce the removed/added hunk embedding into removed/added file embedding. After that, CC2Vec applies element-wise comparison functions to these two file embeddings, deriving the file-level code change representation.

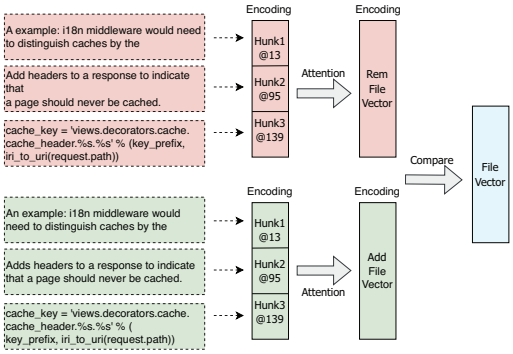

Fig. 2: **How CC2Vec deals with diff in a file.**

While the attention mechanism can effectively model the relations between input sequences (hunks within a modified file) [19], it treats each input hunk as an independent token and neglects the relative position of hunk within the modified file [43], resulting that the yielded file encoding lacks positional information associated with the modified hunks. The comparison functions CC2Vec employed are all element-wise, which operate each corresponding element at identical positions in the removed and added file embeddings. Due to the absence of position information in the reduced file embeddings, element-wise comparison at the file granularity may lead to unnecessary noise. Following the rationale of CC2Vec, when comparing the removed line from hunk@13 (hunk that begins at line 13 in Figure 1) with the added line from hunk@139, it becomes challenging to derive a meaningful semantic. However, when comparing the removed line and added line within hunk@13, it is easy to infer that the semantic meaning of this hunk modification is "fix a typo".

To tackle this challenge, we adopt a hunk-level granularity comparison to capture the semantics of hunk modifications, as depicted in Figure 3. By conducting comparisons at this fine-grained level, we can obtain representations for each hunk that capture the differences between the old and new versions locally. As shown in Figure 3, it is easy to infer that the Hunk Vector @13 indicates the semantic "fix a typo" and the Hunk Vector @139 indicates the semantic "reformat code". Subsequently, we employ transformers [43] to compute the global representation of changes, i.e., the representation for changed files, taking into consideration the interconnected semantics from multiple local hunk changes.

B. Limitation 2: Key Context Information Missing

The second challenge in CC2Vec stems from its lack of consideration for the context of changes, the hunk context and modified files in particular, which are essential for commit classification.

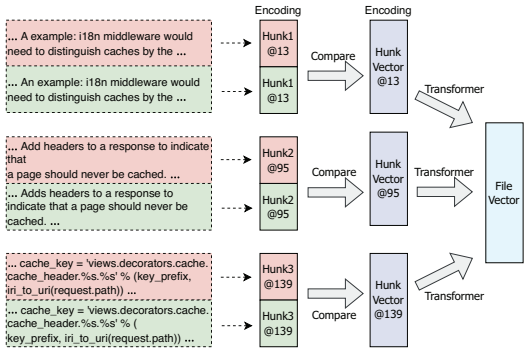

Fig. 3: **How our approach deals with file changes**

Hunk Context refers to the lines that enclose the removed or added lines within a modified hunk, as depicted by the white background region in Figure 1. While the removed and added lines are crucial for commit classification, the context lines can provide additional information, such as the name of the modified methods. The context lines can indicate where the changes occurred and shed some light on the reasons for specific modifications. For instance, in Figure 1, the context of hunk@95 indicates that it represents a simplification of the comments inside a function. This supplementary context helps comprehend the nature and intent of the changes made. Therefore, we deem hunk context necessary in commit classification.

File Category Context refers to the categories of the modified file within the project. In software projects, developers organize software artifacts into different folders [51]. The categorization of files is rich in information for determining the types of maintenance activity. For instance, consider a single file with the path *"tests/gis_tests/gdal_tests/test_raster.py"* that undergoes modification. It can be readily inferred that this file is a test file. As a result, any modifications made to this specific file would likely be classified as changes associated with software testing. Considering the role of software testing in ensuring software quality and functionality [24], such a modification is more likely to be classified as an activity that enhances the software, i.e., 'perfective' following Swanson's definition of maintenance activities [40].

Both the context of modified hunks and files are essential for commit classification. However, CC2Vec neglects these two crucial pieces of context information, focusing only on the added and removed lines in modified files. To address this limitation, our proposed approach fully considers the two contexts. During our data preprocessing phase, we extend the modified lines within hunks to include their contextual lines to form the old and new versions of the hunk. This enables the model to consider the context when comparing the old and new hunks, as shown in Figure 3, where "..." represents the hunk context (details in Section III-A1). To utilize the context of modified files, we categorize the modified files into five distinct categories: source code, build/configuration management, testing, documentation, and others, following the

categorization employed by Hindle et al. [18]. We use the fine-grained line of code (LOC) features to measure the scale of each type of modification and incorporate these change features into our model. Detailed implementation can be found in Section III-C.

III. Model Architecture

COLARE has three components: the diff embedding module, the message embedding module, and the file category fusing module. Figure 4 shows the architecture of COLARE. We provide the details of the three modules in the following sections.

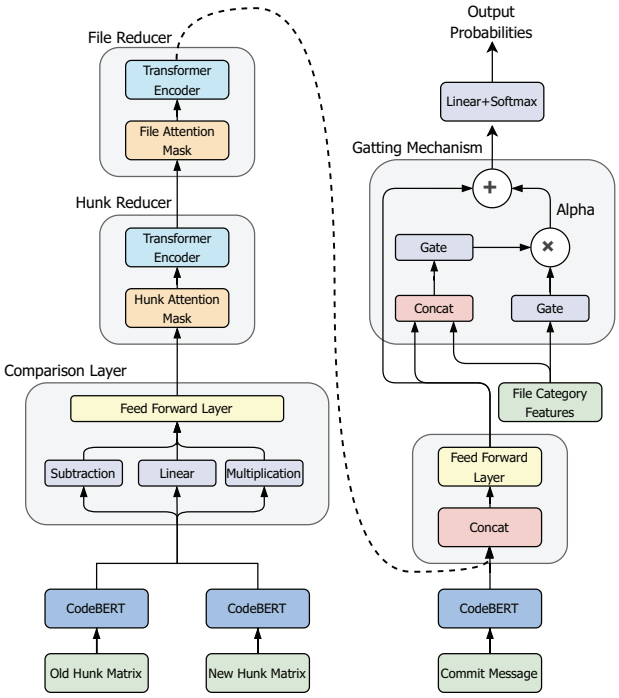

Fig. 4: **Architecture of the proposed model COLARE**

A. *Diff Embedding Module*

This module encodes the changes within commits into effective vector representations. We first perform the **Diff Preprocess** to organize the commit diff into matrices of old and new hunks based on its hierarchical structure. Subsequently, we leverage the pre-trained model CodeBERT [10] to encode these hunks to obtain the **Hunk Representations**. Then, we apply a **Comparison layer** between the old and new hunk encodings to capture the semantics of hunk modifications. In the last, we employ a **Hierarchical Reduction** to compute the ultimate embedding of commit diffs.

*1) **Diff Preprocess**:* As illustrated in the motivation section II-A, we extract old and new code segments within commit diffs at the level of hunk granularity. Specifically, we extract the added and removed code lines within each hunk to form the new and old versions, respectively. Notably, we preserve the default contextual lines from git during extraction to provide context for the modifications in each hunk.

A standard commit diff demonstrates a hierarchical organization, wherein there are one or more modified files, and each file contains one or more hunks consisting of added and removed lines of code accompanied by contextual lines. To effectively capture and retain this hierarchical structural information, we represent the old and new version of hunks in each commit as a two-dimensional matrix denoted by $\mathcal{B} \in \mathbb{R}^{\mathcal{F}\times\mathcal{H}}$, where $\mathcal{F}$ represents the number of modified files, and $\mathcal{H}$ represents the number of modified hunks within each file. Specifically, we use B_o and B_n to represent the two-dimensional matrices of the old and new hunks.

The count of modified files in different commits fluctuates, just like the hunk numbers within each modified file. For parallelization [25], similar to CC2Vec [19], the number of files in each commit is padded or truncated to a hyperparameter $\mathcal{F}$, while the number of hunks in each file is padded or truncated to a hyperparameter $\mathcal{H}$. In contrast to CC2Vec [19], where the padded instances are incorporated into the attention computation, we adopt an attention mask [43] to prevent irrelevant padded instances from participating in the computations. Specifically, during the construction of $\mathcal{B}_o$ and $\mathcal{B}_n$, we also compute file attention mask $\mathcal{M}_f$ ($\mathcal{M}_f \in \{0,1\}^{\mathcal{F}}$) and hunk attention mask $\mathcal{M}_h$ ($\mathcal{M}_h \in \{0,1\}^{\mathcal{F}\times\mathcal{H}}$).

*2) **Hunk Representation**:* To obtain effective hunk representation, we fine-tune a Transformer-based pre-trained language model CodeBERT [10]. CodeBERT has undergone pretraining on a comprehensive dataset encompassing both natural language (NL) and programming language (PL), rendering it well-suited for code understanding tasks [10]. Its effectiveness has been demonstrated in the latest code understanding tasks, such as security patch detection [50]. For a hunk code segment, we first tokenize it using the CodeBERT tokenizer [10] into tokens, then feed these tokens into CodeBERT to obtain feature vector representation. For the classification task, we select the feature vector associated with the special token `[CLS]` as the representation for the code segment. This process can be described by the following equation:

$$\mathcal{D}_o[i][j] = \text{CodeBERT}(\text{Tokenizer}(\mathcal{B}_o[i][j]))$$
$$\mathcal{D}_n[i][j] = \text{CodeBERT}(\text{Tokenizer}(\mathcal{B}_n[i][j]))$$

Here, $\mathcal{D}$ is a matrix with a shape of $\mathcal{F} \times \mathcal{H} \times 768$, where $\mathcal{D}_n[i][j]$ represents the new version of the jth hunk in the ith file in a commit, while $\mathcal{D}_o[i][j]$ denotes the corresponding old version.

*3) **Comparison Layer**:* The semantics of hunk modifications are derived by comparing the old and new versions of the hunk code. For example, by comparing the old and new versions of the hunk@13 shown in Figure 1, we can infer that the semantic of this hunk modification is "fix a typo". To capture this semantic information at the hunk level, we employ three widely adopted comparison functions [19, 44] to compare the old and new hunk encodings: element-wise subtraction, element-wise multiplication, and a feed-forward neural network.

Specifically, for any i ($1 \leq i \leq \mathcal{F}$) and any j ($1 \leq j \leq \mathcal{H}$) representing jth hunk in ith file, we denote the old version of the hunk encoding as $\mathcal{V}_o$ ($\mathcal{V}_o = \mathcal{D}_o[i][j]$) and the new version of the hunk encoding as $\mathcal{V}_n$ ($\mathcal{V}_n = \mathcal{D}_n[i][j]$). For element-wise subtraction, we simply subtract the old version of hunk encoding from the new version of hunk encoding, i.e., $\mathcal{V}_{sub} = \mathcal{V}_n - \mathcal{V}_o$. For element-wise multiplication, we perform an element-wise multiplication between the new version hunk encoding and the old version hunk encoding, i.e., $\mathcal{V}_{mul} = \mathcal{V}_n \odot \mathcal{V}_o$. For the forward neural network, we concatenate the new version hunk encoding and the old version hunk encoding and feed them as input into a forward neural network, i.e., $\mathcal{V}_{nn} = \text{Linear}(\text{ReLU}(\text{Linear}(\mathcal{V}_n \oplus \mathcal{V}_o)))$. Finally, we concatenate the results of the three comparison functions and map the output of the comparison layer to an appropriate dimension using a feed-forward layer. This process can be described by the following equations:

$$\mathcal{V}_{concat} = \text{LayerNorm}(\mathcal{V}_{sub} \oplus \mathcal{V}_{mul} \oplus \mathcal{V}_{nn})$$

$$\mathcal{V}_{compare} = \text{Linear}(\text{ReLU}(\text{Linear}(\mathcal{V}_{concat})))$$

We perform this comparison process on each old and new hunk encoding pair to obtain $\mathcal{X}$, where $\mathcal{X}[i][j]$ represents the semantics of the modification in the jth hunk of the ith file in a commit.

$$\mathcal{X}[i][j] = \text{ComparisonLayer}(\mathcal{D}_o[i][j], \mathcal{D}_n[i][j])$$

*4) **Hierarchical Reduction**:* To leverage the hierarchical structure of commits, wherein one or more files are modified, and each modified file contains one or more modified hunks, we employ a Hierarchical Reduction technique to compute the final embedding of commit diffs from hunk embeddings.

Hunk Reducer takes multiple hunk embeddings as inputs and reduces them to a single file embedding. This process is achieved by employing a Transformer encoder [43], leveraging multi-head attention and feed-forward neural networks. This architecture enables the model to effectively capture the intricate relationships among different positions within the input sequence [43], i.e., the hunk embeddings. To address the impact of padded hunks on attention weights, we introduce the hunk attention mask $\mathcal{M}_h$ to mask the padded hunks, as described in Sec. III-A1.

File Reducer takes multiple file embeddings as inputs and reduces them to a single commit embedding. The architecture of the file reducer mirrors that of the hunk reducer, with the utilization of the file attention mask $\mathcal{M}_f$, as described in Sec. III-A1, to appropriately handle padded file embeddings.

B. Commit Message Embedding Module

CodeBERT [10] is trained on both natural language (NL) and programming language (PL), making it well-suited for the NL-PL understanding task. Similar to commit diff embedding, we also utilize CodeBERT to encode commit messages. Firstly, we employ the CodeBERT tokenizer to tokenize the commit message into a sequence of tokens, which is then fed into CodeBERT. For classification tasks, we choose the output encoding corresponding to the special token `[CLS]` as the embedding for the entire commit message. Furthermore, we concatenate the embedding vectors of the code diff and the commit message, feeding them to a feed-forward layer for dimension mapping. This process results in the generation of an embedding for a commit.

C. File Category Fusing Module

As illustrated in Sec. II-B, the categories of modified files are essential to determine the maintenance activities of commits. In our approach, we propose the file category fusing module to boost the effectiveness of commit classification. Following the file category scheme proposed by Hindle et al. [18], we employ heuristic matching to classify each modified file within a commit into one of five categories: source code, testing, build/configuration management, documentation, and others. Different from the study of Hindle et al. [18] that only considers the number of changed files in each file category, we leverage the file category feature in a more fine-grained way by additionally calculating the number of modified lines in each file category. Since the changed lines in code or non-code files have different impacts on determining the maintenance activity of a commit, we further categorize files into code or non-code types using heuristic suffix matching. We analyze the variations in code lines and comment lines separately for code files. The code lines encapsulate the core functionality of a code file, while the comment lines, which are devoid of functionality, are utilized for explanatory purposes and to convey necessary information. For non-code files, we exclusively monitor changes in line counts. This method allows us to derive the LOC (Lines of Code) features for each file category, thus allowing us to measure the extent of modifications within each category.

Given that the output of Section III-B is the BERT embedding of commits, and the file category features are numerical, these two features might not occupy the same dimensional space. We therefore view these as two distinct modalities and employ the multimodal adaptation gate introduced by Rahman et al. [35] to incorporate our refined file category features into the proposed model. We denote the fusion representation of code diff and commit message as V_{commit}, and our refined file category features as V_{fcat}. The multimodal adaptation gate can then be expressed by the following equation:

$$V_{out} = V_{commit} + \alpha h$$

$$h = g_f \cdot (W_f V_{fcat}) + b_h$$

$$\alpha = \min(\frac{||V_{commit}||_2}{||h||_2} \times \beta, 1)$$

$$g_f = ReLU(W_{gf}[V_{commit} \oplus V_{fcat}] + b_f)$$

Here, $\oplus$ signifies concatenation, V_{out} represents the fused vector representation, which is the final representation of a commit, W_f and W_{gf} are learnable parameters, and β is a hyperparameter.

IV. Experiment Setup

A. Research Questions

In this paper, we aim to answer the following three research questions:

- **RQ1:** How does our model perform compared to the state-of-the-art commit classification technique in the Java dataset?
- **RQ2:** How does our model perform in the multiple programming language scenario?
- **RQ3:** How effective is each component of our model?

The approach presented by Ghadhab et al. [13] currently achieves state-of-the-art performance but is limited to the Java programming language. In **RQ1**, we compare our model against theirs using their Java dataset for fairness. In **RQ2**, we further investigate the performance of our model in a multiple programming language scenario by comparing COLARE with several baselines as well as the zero-shot ChatGPT [33]. In **RQ3**, we perform an ablation study to explore the effectiveness of each component of our model.

B. Datasets

We evaluated our model on two datasets. The first dataset is provided by Ghadhab et al. [13], which is one of our baselines. Comprising a total of $1,793$ labeled commits, this dataset was collected from studies conducted by Levin and Yehudai [28], Amor et al. [2], and Mauczka et al. [31] to ensure a balanced distribution of maintenance activity labels. The second dataset, which we curated and labeled, is a multi-language dataset containing $1,581$ commits from 107 distinct projects written in seven programming languages. We use this dataset to further assess our model's performance in classifying commits in multiple programming languages.

In the realm of automatic commit maintenance classification, the majority of existing studies [28, 30, 13, 17] primarily focus on Java projects. Only Sarwar et al. [37] introduced a dataset encompassing multiple programming languages. However, their publicly available dataset[4] lacks the SHA values of the commits, making retrieving information such as code diffs challenging. Hence, we constructed a multi-language dataset through two steps: *data collection* and *data labeling*.

*a) **Data Collection**:* We selected seven influential programming languages for our study: C++, Python, Java, Go, Rust, JavaScript, and TypeScript. Considering the constraint of manual labeling, we determined to collect a set of 250 commits for each programming language. For Java, we randomly extracted 250 commits from the dataset provided by Levin and Yehudai [28], as this dataset has been widely adopted in related studies [13, 30] and already has labels. For the remaining languages, we identified popular repositories primarily written in each respective language. We followed prior work [37] to ensure comprehensive coverage and quality of the dataset. Specifically, we utilized the GitHub Rest API[5] to sort repositories by star count and selected the top 20 repositories for each language, resulting in a total dataset from 120 repositories.

We performed a manual inspection of these 120 repositories, excluding those classified as resource-sharing or tutorial-oriented repositories, such as Python-100-Days[6], as they do not meet the technical criteria for software development projects. Through this cleanup process, we were left with a total of 96 repositories. These repositories span a wide range of software domains, including programming languages, deep learning frameworks, code editors, front-end frameworks, etc. Table I presents the top three projects selected for each language.

TABLE I: **Top three projects on each programming language**

Programming Language	Top Three Projects
C++	tensorflow, electron, terminal
Python	Python, Auto-GPT, youtube-dl
Go	go, kubernetes, gin
Rust	deno, rust, tauri
JavaScript	react, bootstrap, javascript
TypeScript	vue, vscode, TypeScript

Subsequently, we conducted a random sampling of 250 commits for each programming language. To ensure a balanced representation, we performed uniform sampling within the corresponding repositories for each language. For instance, considering the existence of 18 repositories for the Rust language, we randomly selected approximately $\lfloor \frac{250}{18} \rfloor = 13$ commits from each repository. In the end, we obtained a comprehensive dataset comprising a total of $1,750$ commits spanning across 107 projects written in 7 distinct programming languages, where 250 Java commits were pre-labeled [28], leaving us with $1,500$ commits that required manual annotation.

*b) **Data Labeling**:* We classified commits into Swanson's maintenance activities [40], which are widely followed by prior work [13, 14, 37]. Specifically, Swanson categorized three types of commits that conduct different maintenance activities:

1) **Corrective:** fixing bugs and faults in the project.
2) **Perfective:** enhancement of the project, such as performance enhancement and source code refactoring.
3) **Adaptive:** modifications to the project to adapt it to the new environment such as the feature addition.

Regarding the labeling rules, we adhere to the annotation guidelines employed by prior work [37]. Performing multiple types of modifications in a single commit is considered bad practice as it undermines maintainability [1, 38, 4]. Hence, we adopt a multi-class labeling scheme, wherein each commit is assigned one unique label.

The annotation process involved the first and fourth authors of this study. During this process, commits that encompassed multiple changes or were automatically generated, such as "Merge Branch" and "Revert Commit" were excluded. Consequently, we were left with a final set of $1,331$ commits after removing 169 commits. Upon labeling the commits, the two authors assigned different labels to 270 (20%) commits. The

[4]https://zenodo.org/record/3948445

[5]https://docs.github.com/en/rest

[6]https://github.com/jackfrued/Python-100-Days

resulting Cohen's kappa coefficient [6] between the two raters was 0.55, indicating a moderate level of agreement [26]. To address the discrepancies, three meetings were conducted to discuss these 270 cases. When the two authors could not reach a consensus, a third author served as an arbitrator.

In the end, we obtained a total of $1,331$ commits with maintenance activity labeled. Additionally, by incorporating 250 randomly sampled Java commits from the dataset provided by Levin and Yehudai [28], our final dataset comprised $1,581$ commit records. Among these, there were 989 commits labeled as Perfective, 353 commits labeled as Corrective, and 239 commits labeled as Adaptive. Considering the approach taken for data collection, this label distribution aligns with proportions typically observed in real-world software projects.

C. Compared Techniques

In this study, we conducted a comprehensive comparative analysis of COLARE against three baseline methods, along with the zero-shot ChatGPT [33].

Ghadhab et al. [13] achieved state-of-the-art results by leveraging fine-grained source code changes and BERT [8] to enhance commit classification. The fine-grained source code change features are mined through tools such as ChangeDistiller [12]. The applicability of their approach, however, is constrained to Java, as it is the only language supported by the code distillers they employed [13]. Therefore, we compared COLARE with their method on their dataset.

Sarwar et al. [37] fine-tuned a DistilBERT model for commit classification, and Lee et al. [27] fine-tuned both CodeBERT and RoBERTa to handle code diffs and commit messages, respectively. These methods are designed to be language-independent, which motivated us to compare COLARE with their approaches on our multi-language dataset. Furthermore, the Large Language Model-based ChatGPT [33] has recently demonstrated remarkable achievements in code understanding tasks, such as code summarization [41]. Considering this noteworthy performance and popularity of ChatGPT, we extended our evaluation to assess COLARE's performance in comparison with ChatGPT.

To ensure the validity of our comparison, we reused the implementations of the compared techniques from their reproducible packages. In the case of [27], where their packages were unavailable, we re-implement the techniques strictly following the description in their paper.

D. Implementation Details

Due to the constraints imposed by GPU memory capacity, it is observed that in our dataset, approximately 80% of commits comprise less than 4 modified files, and around 88% of files contain fewer than 3 modified hunks. Consequently, in the commit diff embedding module, we have configured the hyperparameter $\mathcal{F}$ to be set to 4 and the hyperparameter $\mathcal{H}$ to 3. Additionally, the hidden layer size of the code representation, which corresponds to the output dimension of the comparison layer, is set to 128. For computing the embeddings of files and commits, we employ a two-layer Transformer Encoder with a dropout rate [39] of 0.15.

In the commit embedding module, we utilize the pre-trained CodeBERT [10] model with hidden size 768 provided by the Hugging Face Transformer library. The weights of CodeBERT, which are used for embedding commits, are shared with the weights of CodeBERT in the code diff embedding module. When combining the embeddings of the code diff and the commit message, we set the output dimension of the feed-forward layer to 768 and the hidden layer size to 2048.

In the file category fusing module, we utilize the multi-adaptation gate mechanism [35] implemented in the multi-transformers library developed by Gu et al. [15], with a dropout rate [39] of 0.15. The hyperparameter of the multi-adaptation gate mechanism is the default to 0.2.

We use cross-entropy loss function and AdamW optimizer [29]. Considering that the lower layers of the BERT model may contain more general information, we adopt a layer-wise learning rate strategy for CodeBERT, following [22]. Specifically, the learning rate for the first three layers of BERT is set to $1e^{-5}$; for the next four layers it is set to $2e^{-5}$, and for the final four layers it is set to $4e^{-5}$. The learning rates for the remaining modules are all set to $1e^{-4}$.

All experiments were performed on a server with two 24G NVIDIA GeForce RTX 3090 GPUs, an Intel(R) Xeon(R) Silver 4310 CPU, and the Ubuntu 20.04.2 LTS operating system.

E. Evaluation Metrics

Given that we perform multi-class classification on commits, following the footsteps of prior research in commit classification [32, 2, 18, 28, 14, 37, 13, 20, 30, 27], we employ the widely-used metrics: accuracy, precision, recall, and F1 score. Moreover, considering our task involves classifying commits into three distinct classes, we also incorporate macro average metrics [21] into our evaluation framework to facilitate a more comprehensive assessment. The macro average is computed as the arithmetic mean of all the per-class metrics, i.e., precision, recall, and F1 score.

V. Results

A. RQ1: Comparison against Baseline in Java

a) ***Method****:* We conducted a comparison between our proposed method, COLARE, and the approach proposed by Ghadhab et al. [13] (denoted as *GClassifier*), which currently stands as the state-of-the-art multi-class commit classification method. As mentioned in Section III-A3, we compare COLARE's performance against *GClassifier* using the dataset Ghadhab et al. provided. During the evaluation, we found that the SHAs of 12 commits in their dataset are missing, making retrieving the corresponding code diffs difficult. Therefore, we removed the 12 commits, and the remaining $1,781$ commits are used for comparison.

Our evaluation methodology employed a stratified five-fold cross-validation technique [5] to comprehensively assess the performance of the model. This approach randomly divided the dataset into five subsets, each maintaining the same label

distribution as the original dataset. In each iteration, one subset was designated as the test set, while the remaining four subsets served as the training set. This partitioning process was uniformly applied for both *GClassifier* and COLARE. The evaluated models were trained on the training sets and tested on the test set. We report the average metrics obtained from the five testing iterations.

TABLE II: **Classification results of COLARE and baseline (stratified five-fold cross-validation)**

Metrics	GClassifier[13]	COLARE
$\text{Precision}_{\text{corrective}}$	72.50%	**82.77%**
$\text{Recall}_{\text{corrective}}$	74.50%	**81.53%**
$\text{F1}_{\text{corrective}}$	73.20%	**82.09%**
$\text{Precision}_{\text{perfective}}$	75.49%	**85.02%**
$\text{Recall}_{\text{perfective}}$	70.00%	**79.61%**
$\text{F1}_{\text{perfective}}$	72.25%	**82.05%**
$\text{Precision}_{\text{adaptive}}$	78.08%	**78.55%**
$\text{Recall}_{\text{adaptive}}$	79.11%	**84.36%**
$\text{F1}_{\text{adaptive}}$	77.93%	**81.30%**
Macro precision	75.36%	**82.11%**
Macro Recall	74.53%	**81.83%**
Macro F1	74.46%	**81.81%**
Accuracy	74.57%	**81.81%**

*b) **Results**:* Table II showcases the classification results of COLARE and *GClassifier* approach, with the superior results highlighted in **bold**. We can see that COLARE consistently outperforms *GClassifier* across all metrics for commit classification. Specifically, COLARE surpasses the baseline by 7.24% in terms of accuracy. Furthermore, COLARE's performance betters the baseline in macro precision, macro recall, and macro F1 score by 6.75%, 7.30%, and 7.35%, respectively. When considering different maintenance activity classes, COLARE also demonstrates significant improvements compared to the baseline, especially in the 'Perfective' class, where the improvement of the F1 score is nearly 10%.

Summary for RQ1: COLARE outperforms the state-of-the-art baseline for automatic commit maintenance classification on the Java dataset, with improvements of 7.24% and 7.35% in accuracy and macro F1 score, respectively.

B. RQ2: Effectiveness of COLARE in a Multi-language Scenario

*a) **Method**:* In this research question, we delve further into evaluating the performance of COLARE across projects involving multiple programming languages. We compare COLARE against two baseline approaches: Sarwar et al. [37] (denoted as *SClassifier*) and Lee et al. [27] (denoted as *LClassifier*). Recognizing the remarkable performance of ChatGPT [33] in a variety of code-related tasks [41], we also benchmark our model against ChatGPT. Given that ChatGPT is driven by prompts [45], we adopted the Input Semantics pattern proposed by White et al. [45] to build our prompt for ChatGPT. Detailed information regarding the prompt can be accessed in our online appendix [47]. This carefully curated prompt encapsulates the classification task and the meaning of each maintenance activity and requests ChatGPT to assign a label for a particular commit. We use this prompt to send each commit's message and code diff to ChatGPT3.5 [33], which in turn generates a maintenance activity label for each commit.

We conduced the experiments on the multi-language dataset annotated in Sec. IV-B. Similar to the method adopted in RQ1, we apply stratified five-fold cross-validation to evaluate the models and report the average performance.

TABLE III: **Classification results of COLARE and the three baselines (stratified five-fold cross-validation)**

Metrics	ChatGPT	SClassifier	LClassifier	COLARE
$\text{Precision}_{\text{corrective}}$	49.68%	81.98%	**86.55%**	85.40%
$\text{Recall}_{\text{corrective}}$	90.08%	84.53%	82.51%	**91.10%**
$\text{F1}_{\text{corrective}}$	64.00%	83.20%	84.37%	**88.03%**
$\text{Precision}_{\text{perfective}}$	**82.47%**	67.80%	69.50%	80.84%
$\text{Recall}_{\text{perfective}}$	32.56%	65.18%	**78.44%**	72.83%
$\text{F1}_{\text{perfective}}$	46.55%	66.33%	73.17%	**76.31%**
$\text{Precision}_{\text{adaptive}}$	28.58%	61.86%	67.21%	**80.72%**
$\text{Recall}_{\text{adaptive}}$	65.22%	56.50%	63.17%	**66.93%**
$\text{F1}_{\text{adaptive}}$	39.64%	58.59%	64.38%	**72.33%**
Macro precision	53.58%	70.55%	74.42%	**82.32%**
Macro Recall	62.62%	68.74%	74.70%	**76.96%**
Macro F1	50.06%	69.37%	73.98%	**78.89%**
Accuracy	50.35%	75.97%	78.68%	**83.37%**

*b) **Results**:* Table III presents the outcomes of our comparative analysis, with the highest scores highlighted in **bold**. As evidenced by the table, COLARE surpasses all the compared methods, demonstrating superior performance in both accuracy and macro F1 score. The improvements achieved by COLARE span from 4.69% to 33.02% in accuracy and from 4.9% to 28.83% in macro F1 score. These improvements underscore the effectiveness and generalization capabilities of COLARE. In regard to the corrective and perfective classes, ChatGPT and *LClassifier* exhibit the highest performance in some specific metrics, indicating potential directions for future work to further enhance the performance of these particular maintenance classes.

Summary for RQ2: In the context of multiple programming languages, COLARE outperforms all baseline approaches, showcasing improvements that range from 4.69% to 33.02% in accuracy and from 4.9% to 28.83% in macro F1 score, respectively.

C. RQ3: Ablation Study

*a) **Method**:* In this research question, we first conduct an ablation study to investigate whether our adapts on the commit embedding module are effective. Since we optimized CC2Vec to encode commit diff for commit maintenance activity classification, we constructed a variant model denoted as $\text{COLARE}_{\text{cc2vec}}$, whereby we substituted the commit diff embedding module with CC2Vec module.

Furthermore, we delve deeper into the effectiveness of the three key components in COLARE's design: (1) Commit diff embedding module, which captures changes at the hunk level and leverages the hierarchical structure of commits to encode commit diffs. (2) Commit message embedding module, which utilizes a pre-trained CodeBERT to encode commit messages. (3) File category fusing module, which integrates fine-grained file category features using a multimodal adaptation gate to enhance model performance. To investigate the effectiveness of these three key components, we construct three variants of COLARE: COLARE$_{\text{-code}}$, COLARE$_{\text{-msg}}$, and COLARE$_{\text{-fcat}}$. Specifically, COLARE$_{\text{-code}}$ removes the commit diff embedding module. COLARE$_{\text{-msg}}$ removes the commit message embedding module and directly combines the commit diff encoding with fine-grained file category features for classification. COLARE$_{\text{-fcat}}$ eliminates the fine-grained file category features and directly combines the commit diff embedding with the commit message embedding for classification. For evaluation, we partitioned the dataset into training, validation, and test datasets (7:1:2 ratio). We trained each model for 10 epochs, selected the best-performing model based on the validation set, and reported the evaluation metrics on the test set.

TABLE IV: **Performance comparison between COLARE$_{\text{cc2vec}}$ and COLARE**

Metrics	COLARE$_{\text{cc2vec}}$	COLARE
Accuracy	80.76%	**87.06%**
Macro F1	72.71%	**85.08%**

TABLE V: **Results of the ablation study**

Metrics	COLARE$_{\text{-code}}$	COLARE$_{\text{-msg}}$	COLARE$_{\text{-fcat}}$	COLARE
Accuracy	82.97%	66.56%	85.80%	**87.06%**
Macro F1	79.40%	56.47%	82.08%	**85.08%**

*b) **Results**:* Table IV presents the performance comparison between the COLARE and COLARE$_{\text{cc2vec}}$ models. COLARE outperforms COLARE$_{\text{cc2vec}}$ in both accuracy and macro F1 score, indicating that our adaptations, particularly the comparison of code changes at the hunk level, indeed enhance the performance of maintenance activity classification. Table V showcases the effectiveness of the standard COLARE model and its variants. COLARE achieves superior performance in both accuracy and macro F1 score compared to its variants. COLARE$_{\text{-code}}$ exhibits a drop in performance, with a decrease of 4.09% in accuracy and 5.68% in macro F1 score when compared to COLARE. Similarly, when the file category fusing module is removed, COLARE$_{\text{-fcat}}$ demonstrates a reduction in performance for both accuracy and macro F1 score. These results suggest that involving commit diff and file category information plays a positive role in commit classification. The variant COLARE$_{\text{-msg}}$ exhibits a significant performance decline compared to COLARE$_{\text{-code}}$ and COLARE$_{\text{-fcat}}$. This decrease underscores the significance of the commit message in maintenance activity classification.

Summary for RQ3: The three key design components in COLARE, i.e., the hierarchical pre-trained commit diff embedding module, commit message embedding module, and fine-grained file category features, are all effective and contribute to the improvement of COLARE's performance. The comparison with COLARE$_{\text{cc2vec}}$ demonstrates the significance of addressing the coarse-grained comparison limitation of CC2Vec.

VI. Discussion

This section discusses the potential application of our proposed approach and the validity threats of this study.

A. Conventional Commit Classification

Our approach is designed to classify commits into three key maintenance activities. Recently, increasing projects have applied conventional standard [7] to categorize commit maintenance activity. Conventional commits mandate developers to adhere to a well-defined set of rules when crafting commit messages, necessitating the inclusion of a specific type for each commit. The maintenance types of conventional commits encompass ten distinct categories, which refine the 'perfective' in Swanson's commit maintenance categories [40] into subcategories like test, ci, docs, refactor, and others. Despite evaluating our model in the three maintenance activities, which are widely used in existing classification studies, we also explore whether our model is still effective in conventional commit classification.

To collect commits that conform to the rules of conventional commits, we selected the repository acquired from Sec. IV-B0a, comprising a total of 96 repositories written in 6 programming languages. Utilizing regular expressions, we parsed and extracted commits that met the criteria for conventional commits, resulting in a total of $116,292$ qualifying commits. To control the training time in an appropriate frame, we performed a random sampling of $50,000$ commits from this pool. We divided the dataset into training, validation, and test sets in an 8:1:1 ratio. After evaluation, our proposed model, COLARE, achieved 70.70% accuracy, 60.28% macro F1, and 70.11% weighted F1. To demonstrate the effectiveness of COLARE, we also implemented one baseline [8] to classify commits into conventional types and achieved 66.72% accuracy, 58.60% macro F1, and 66.56% weighted F1. The 3.98% improvements in accuracy, 1.68% improvements in macro F1, and 3.55% improvements in weighted F1 illustrate the effectiveness of COLARE when applied in the conventional scenario.

B. Practical Application

The automatic commit classification for maintenance activities facilitates the establishment of a well-organized tracking system for development history, yielding various benefits for software development. To promote the application of COLARE, we have created a web dashboard to host our trained model (demo accessible at [47]). We developed a user-friendly

interface that enables users to input any GitHub repository URL or commit URL to obtain the classification labels. Users can simply input their commit data to COLARE via the dashboard, and then the maintenance activities of these commits will be displayed. Software practitioners can directly use the categorized commits to improve work efficiency and quality. For instance, software managers can gain a comprehensive understanding of a repository's maintenance activities and make informed decisions. Researchers can take a further step in measuring the capability and expertise of developers in a finer granularity.

C. Threats to Validity

The section discusses potential threats to the validity. The first threat lies in the multiple programming language dataset we constructed. Sampling commits from popular repositories across various programming languages may result in a limited representation of less active projects. However, We deem the impact of this threat to be trifling since our sampling approach aligns with previous work [37], and our sample covers these projects' complete lifecycles, which contain their initial and inactive stages. Additionally, while the selection of 250 commits for each language might not represent the optimal strategy for dataset creation, we had to balance the volume of each language against the constraints imposed by the cost of manual annotation. As a result, the size of our final dataset is broadly in line with those used in previous studies [28, 13]. Furthermore, the manual labeling process may introduce subjectivity due to different raters. To mitigate this concern, two authors independently labeled the data, and we conducted several meetings to resolve any discrepancies.

The second threat relates to baseline implementation. As the evaluation dataset changed, we took measures to replicate the baseline models. For this purpose, we directly reused the implementation of the baselines from their reproducible packages whenever available. In the case of [27], where the replication package was not made available, we re-implemented their technique meticulously following the details provided in their paper [27]. Given the implementation of Ghadhab et al.'s [13] slightly lower than their reported performance, we contacted the authors of [13] and obtained their recognition.

Finally, in Sec. IV-D, GPU memory limitations led to setting hyperparameters $\mathcal{F}$ and $\mathcal{H}$ to 4 and 3, respectively. With more robust GPUs, these parameters can be increased, potentially reducing truncations and enhancing COLARE's performance.

VII. Related Work

With the increasing complexity of software development, classifying code changes into different maintenance activities has been widely explored with the aim of easing the burden of software management. The commit message, which developers use to describe the changes they made within a repository, serves as a crucial role in all commit maintenance activity classification studies. In these studies, Mockus et al. [32] and Hindle et al. [18] employed word frequency analysis to select keywords from commit messages as features. Levin and Yehudai [28] utilized one-hot keyword vectors extracted from commit messages as features, which were further extended by Herivčko et al. [16] through the exploitation of semantic similarities between words in the commit message. Gharbi et al. [14] represented commit messages using the Term Frequency-Inverse Document Frequency (TF-IDF) technique. Heričko et al. [17] leveraged word2vec to represent words in the commit message and aggregated the word vectors into a commit message vector. Sarwar et al. [37], Ghadhab et al. [13], and Lee et al. [27] employed Transformers [43] based techniques such as BERT [8] to encode and model commit messages. These methods face a major challenge: their reliance on commit message quality, which is often subpar. Dyer et al. [9] reported over 14% of SourceForge project commit messages were empty. Tian et al. [42] found nearly 44% of messages in active open-source projects needed improvement.

To mitigate this issue, several studies have explored features of code diffs within commits. For instance, Levin and Yehudai [28] used ChangeDistiller [12] to extract features from code changes, such as the frequency of condition expression modifications. Building upon Levin and Yehudai's [28] code features, Mariano et al. [30] expanded the feature set to include attributes such as the number of modified files and added lines of code. Hönel et al. [20] utilized various methods to analyze the code density of commits for classification purposes. Popoola et al. [34] employed Random Forest to classify commits based on seven features extracted from code changes. Ghadhab et al. [13] combined bug fixes and refactoring-related features from code diffs with features extracted using BERT for commit classification, achieving state-of-the-art performance. Lee et al. [27] utilized CodeBERT [10] to directly encode code diffs, aiming to enhance the performance of identifying vulnerability-related commits. In this study, we introduce an enhanced code change representation technique to encode commit diffs, integrating commit message embeddings and detailed file category features. Our approach has achieved the best performance in commit classification.

VIII. Conclusion

In this study, we optimized a code changes representation model, CC2Vec, to classify commits into three maintenance activities. We introduced three key improvements to CC2Vec: utilizing CodeBERT for code encoding, doing hunk-level comparison, and incorporating fine-grained features of changed files and code lines. Our evaluations were conducted on both Java and multiple programming language datasets, revealing that our approach surpasses the state-of-the-art, including ChatGPT 3.5. To facilitate replications or future work, we provide the data, scripts, and other resources used in this study at: https://zenodo.org/records/10500219.

IX. Acknowledgment

This work is supported by the National Natural Science Foundation of China Grant (62141209, 62202048, and 62232003) and the Open Fund of National Key Laboratory of Parallel and Distributed Computing (PDL).

References

[1] Kapil Agrawal, Sadika Amreen, and Audris Mockus. Commit quality in five high performance computing projects. In *2015 IEEE/ACM 1st International Workshop on Software Engineering for High Performance Computing in Science*, pages 24–29. IEEE, 2015.

[2] Juan Jose Amor, Gregorio Robles, Jesus M Gonzalez-Barahona, and Alvaro Navarro. Discriminating development activities in versioning systems: A case study. In *Proceedings PROMISE*, volume 2006, page 2nd, 2006.

[3] Sadika Amreen, Andrey Karnauch, and Audris Mockus. Developer reputation estimator (dre). In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1082–1085. IEEE, 2019.

[4] CHRIS BEAMS. How to write a git commit message. https://chris.beams.io/posts/git-commit/, 2014.

[5] Daniel Berrar et al. Cross-validation., 2019.

[6] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20 (1):37–46, 1960.

[7] Conventional-Commits. Conventional commits. https://www.conventionalcommits.org/en/v1.0.0/, 2014.

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *North American Chapter of the Association for Computational Linguistics*, 2019.

[9] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 422–431. IEEE, 2013.

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

[11] Inc Free Software Foundation. Hunks (comparing and merging files). https://www.gnu.org/software/diffutils/manual/html_node/Hunks.html, 2023. Accessed: 2023-7-24.

[12] Harald C Gall, Beat Fluri, and Martin Pinzger. Change analysis with evolizer and changedistiller. *IEEE software*, 26(1):26–33, 2009.

[13] Lobna Ghadhab, Ilyes Jenhani, Mohamed Wiem Mkaouer, and Montassar Ben Messaoud. Augmenting commit classification by using fine-grained source code changes and a pre-trained deep neural language model. *Information and Software Technology*, 135:106566, 2021.

[14] Sirine Gharbi, Mohamed Wiem Mkaouer, Ilyes Jenhani, and Montassar Ben Messaoud. On the classification of software change messages using multi-label active learning. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, pages 1760–1767, 2019.

[15] Ken Gu and Akshay Budhkar. A package for learning on tabular and text data with transformers. In *Proceedings of the Third Workshop on Multimodal Artificial Intelligence*, pages 69–73, 2021.

[16] Tjaša Heričko and Boštjan Šumak. Using domain-specific word embeddings to boost keyword-based commit classification. In *Proceedings of the 13th Conference Data analysis methods for software systems–DAMSS. https://doi.org/10.15388/DAMSS*, volume 13, 2022.

[17] Tjaša Heričko, Saša Brdnik, and Boštjan Šumak. Commit classification into maintenance activities using aggregated semantic word embeddings of software change messages. *Proceedings http://ceur-ws. org ISSN*, 1613:0073, 2022.

[18] Abram Hindle, Daniel M German, Michael W Godfrey, and Richard C Holt. Automatic classication of large changes into maintenance categories. In *2009 IEEE 17th International Conference on Program Comprehension*, pages 30–39. IEEE, 2009.

[19] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. Cc2vec: Distributed representations of code changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 518–529, 2020.

[20] Sebastian Hönel, Morgan Ericsson, Welf Löwe, and Anna Wingkvist. Using source code density to improve the accuracy of automatic commit classification into maintenance activities. *Journal of Systems and Software*, 168:110673, 2020.

[21] Mohammad Hossin and Md Nasir Sulaiman. A review on evaluation metrics for data classification evaluations. *International journal of data mining & knowledge management process*, 5(2):1, 2015.

[22] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. In *Annual Meeting of the Association for Computational Linguistics*, 2018.

[23] Huaxi Jiang, Jie Zhu, Li Yang, Geng Liang, and Chun Zuo. Deeprelease: Language-agnostic release notes generation from pull requests of open-source software. In *2021 28th Asia-Pacific Software Engineering Conference (APSEC)*, pages 101–110. IEEE, 2021.

[24] Mohd Ehmer Khan and Farmeena Khan. Importance of software testing in software development life cycle. *International Journal of Computer Science Issues (IJCSI)*, 11(2):120, 2014.

[25] Yoon Kim. Convolutional neural networks for sentence classification. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, 08 2014. doi: 10.3115/v1/D14-1181.

[26] J Richard Landis and Gary G Koch. The measurement of observer agreement for categorical data. *biometrics*, pages 159–174, 1977.

[27] Jian Yi David Lee and Hai Leong Chieu. Co-training for commit classification. In *Proceedings of the Seventh Workshop on Noisy User-generated Text (W-NUT 2021)*, pages 389–395, 2021.

[28] Stanislav Levin and Amiram Yehudai. Boosting automatic commit classification into maintenance activities by utilizing source code changes. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 97–106, 2017.

[29] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2017.

[30] Richard VR Mariano, Geanderson E dos Santos, and Wladmir Cardoso Brandão. Improve classification of commits maintenance activities with quantitative changes in source code. In *ICEIS (2)*, pages 19–29, 2021.

[31] Andreas Mauczka, Florian Brosch, Christian Schanes, and Thomas Grechenig. Dataset of developer-labeled commit messages. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 490–493. IEEE, 2015.

[32] Mockus and Votta. Identifying reasons for software changes using historic databases. In *Proceedings 2000 International Conference on Software Maintenance*, pages 120–130. IEEE, 2000.

[33] OpenAI. Chatgpt: Optimizing language models for dialogue. https://openai.com, 2022.

[34] Saheed Popoola, Xin Zhao, Jeff Gray, and Antonio Garcia-Dominguez. Classifying changes to models via changeset metrics. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, pages 276–285, 2022.

[35] Wasifur Rahman, Md Kamrul Hasan, Sangwu Lee, Amir Zadeh, Chengfeng Mao, Louis-Philippe Morency, and Ehsan Hoque. Integrating multimodal information in large pretrained transformers. In *Proceedings of the conference. Association for Computational Linguistics. Meeting*, volume 2020, page 2359. NIH Public Access, 2020.

[36] Christian Rodríguez-Bustos and Jairo Aponte. How distributed version control systems impact open source software projects. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 36–39. IEEE, 2012.

[37] Muhammad Usman Sarwar, Sarim Zafar, Mohamed Wiem Mkaouer, Gursimran Singh Walia, and Muhammad Zubair Malik. Multi-label classification of commit messages using transfer learning. In *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 37–42. IEEE, 2020.

[38] Ben Straub Scott Chancon. Pro git. https://git-scm.com/book/en/v2/Distributed-Git-Contributing-to-a-Project, 2014.

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

[40] E Burton Swanson. The dimensions of maintenance. In *Proceedings of the 2nd international conference on Software engineering*, pages 492–497, 1976.

[41] Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F Bissyandé. Is chatgpt the ultimate programming assistant–how far is it? *arXiv preprint arXiv:2304.11938*, 2023.

[42] Yingchen Tian, Yuxia Zhang, Klaas-Jan Stol, Lin Jiang, and Hui Liu. What makes a good commit message? In *Proceedings of the 44th International Conference on Software Engineering*, pages 2389–2401, 2022.

[43] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[44] Shuohang Wang and Jing Jiang. A compare-aggregate model for matching text sequences. *arXiv preprint arXiv:1611.01747*, 2016.

[45] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382*, 2023.

[46] Jianyu Wu, Weiwei Xu, Kai Gao, Jingyue Li, and Minghui Zhou. Characterize software release notes of github projects: Structure, writing style, and content. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 473–484. IEEE, 2023.

[47] Qunhong Zeng, Yuxia Zhang, Zeyu Sun, Yujie Guo, and Hui Liu. Replication package, 2024. URL https://zenodo.org/records/10500219.

[48] Yuxia Zhang, Minghui Zhou, Audris Mockus, and Zhi Jin. Companies' participation in oss development–an empirical study of openstack. *IEEE Transactions on Software Engineering*, 47(10):2242–2259, 2021. doi: 10.1109/TSE.2019.2946156.

[49] Yuxia Zhang, Klaas-Jan Stol, Hui Liu, and Minghui Zhou. Corporate dominance in open source ecosystems: a case study of openstack. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1048–1060, 2022.

[50] Jiayuan Zhou, Michael Pacheco, Zhiyuan Wan, Xin Xia, David Lo, Yuan Wang, and Ahmed E Hassan. Finding a needle in a haystack: Automated mining of silent vulnerability fixes. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 705–716. IEEE, 2021.

[51] Jiaxin Zhu, Minghui Zhou, and Audris Mockus. Patterns of folder use and project popularity: A case study of github repositories. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–4, 2014.