Machine Learning (2025) 114:94
https://doi.org/10.1007/s10994-024-06671-3

Capturing the context-aware code change via dynamic control flow graph for commit message generation

Yali Du[1,2] · **Ying Li**[1,2] · **Yi-Fan Ma**[1,2] · **Ming Li**[1,2]

Received: 27 May 2024 / Revised: 14 August 2024 / Accepted: 13 December 2024 /
Published online: 19 February 2025

Abstract

Commit messages that summarize code changes of each commit in natural language help developers understand code changes without digging into implementation details, thus playing an essential role in comprehending software evolution. In constructing models for automatic commit message generation, prior research has focused on extracting information from the changed code hunks (i.e., code difference), while ignoring the unchanged code hunks (i.e., code context). However, most studies often neglect the fact that the code change is context-aware, that is the semantics of the code difference are heavily dependent on its code context. To take the code context into account, a key challenge arises: the extensive code context may overshadow the minuscule code difference in capturing the changed semantics, which is a disadvantage to commit message generation. In this paper, we propose the dynamic control flow graph (DCFG), which combines both the code contexts and code differences into one dynamic global–local structure. Based on DCFG, we design a novel framework termed capturing the context-aware code change for commit message generation (C^4MG), which attempts to model the changed semantics of the code change based on the relevant code context, while avoiding being misled by the overwhelming amount of unchanged code context. Extensive experiments demonstrate that benefiting from modeling the context-aware code change, C^4MG outperforms not only the state-of-the-art open-source models but also the large language models (e.g., LLaMA3, GPT-4o, and Gemini) on the commit message generation.

Editors: Kee-Eung Kim, Shou-De Lin.

✉ Ming Li
lim@lamda.nju.edu.cn

Yali Du
duyl@lamda.nju.edu.cn

Ying Li
liy@lamda.nju.edu.cn

Yi-Fan Ma
mayf@lamda.nju.edu.cn

[1] National Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China

[2] School of Artificial Intelligence, Nanjing University, Nanjing 210023, China

Keywords Machine learning · Commit message generation · Dynamic control flow graph

1 Introduction

When developers commit changed code to a version control system like Git, each commit is supposed to be documented with a commit message. The commit message summarizes the semantic change of the modified code in natural language, which makes it easy for developers to understand the high-level intention behind the code change without delving into the detailed implementation (Dong et al., 2022). However, manually writing commit messages is time-consuming and labor-intensive. With modern software systems evolving rapidly, although commit messages are vital for software maintenance, developers often neglect the message due to the heavy burden. As the report in SourceForge[1] indicated, around 14% of commit messages in more than 23K open-source Java projects are empty (Dyer et al., 2013). To tackle the ever-growing high cost of writing commit messages, commit message generation has drawn significant attention in the software mining community.

Unlike other code understanding tasks handling one static version of code, commit message generation is required to capture the changed semantics between the pre-changed and post-changed versions. Hence, existing studies mostly focus on modeling the changed code hunks (i.e., code difference), while ignoring the unchanged code hunks (i.e., code context) for commit message generation (He et al., 2023; Wang et al., 2023). For example, CoDisum (Xu et al., 2019) represents the code change by concatenating the flat token sequences of pre-changed and post-changed code differences into one sequence. ATOM (Liu et al., 2020) represents the code changes by concatenating the pre-changed and post-changed paths of the abstract syntax trees (AST). However, in most modification scenarios, the semantics of the changed code snippets have a dependency on the code context. If the code context is discarded and only code differences are retained, the model may fall into the trap of purely memorizing the differences without understanding their semantic meaning (Shi et al., 2019).

Although a few works like (Xu et al., 2022) have attempted to introduce the context as part of the input for commit message generation, the interaction of the code difference and code context in code change learning is still limited by modeling independently before combining and treating as equally important. However, as the code difference is only a tiny fraction of the commit, if all the code context is retained without restriction, the huge context may introduce unnecessary redundancy. For example, in the more than 25K commits of repository tomcat,[2] the changed tokens are no more than 14% of all the tokens. It indicates that the code context is usually overly dominant in one commit in real-world development, and the huge code context as one part of the input without restriction will overwhelm the tiny code difference in the code change learning. This overwhelming misleads the model to focus on the semantics of the code context rather than the changed semantics of the small but critical code difference in the automatic commit message generation.

One question arises here: how to reserve the semantic dependency between the code difference and the code context, while avoiding being misled by the overwhelming amount

[1] https://sourceforge.net/

[2] https://github.com/apache/tomcat.

of the code context in commit message generation? Thus, we design the **D**ynamic **C**ontrol **F**low **G**raph (**DCFG**) to emphasize the local code difference while leveraging the code context to reserve the dependent relationship at the abstract level. The graph combines both the code contexts and code differences into one dynamic global–local structure. In the DCFG, the abstract control flow represents global code including code contexts, and each node of the abstract control flow can be dynamically extended to a block-level control flow. At the block level, the *intra-block* dependency can represent the code semantics of the changed code blocks at a fine-grained level. Meanwhile, at the abstract level, the *inter-block* dependency can ensure context-aware semantic correctness while avoiding redundancy. This block is a compound statement (e.g., for loops, if statements, switch statements) without including other compound statements internally.

As most prior methods mainly focus on the code difference of single granularity, they are hard to directly adopt to capture the changed semantics guided by the global–local DCFG. To this end, a new approach is proposed to capture the context-aware semantics of the code change with the code context at the abstract level, while comparing the code difference at a fine-grained level. Benefiting from the global–local structure information of DCFG, a specific dynamic encoder is designed, including the block embedding module to capture *intra-block* dependency, the pre-context fusion module to capture *pre-changed inter-block* dependency, and the post-context fusion module to capture *post-changed inter-block* dependency, respectively. Moreover, to avoid being misled by the overwhelming amount of the code context, the code differences are amplified by the fine-grained code difference learning module and then inputted into the message decoder to generate commit messages. Additionally, a context invariance constraint is designed to preserve the consistency of code context in the modification, that is the semantics imported by the unchanged code context to the pre-changed block and post-changed block should be consistent.

In our work, we highlight modeling the code difference based on the relevant code context, while avoiding being misled by the overwhelming amount of unchanged code context in commit message generation. To represent the interaction in one commit, we design the DCFG, which is a global–local graph. Benefiting from the control flow paths that reflect the statement-level program behavior, the control flow graph provides a better representation of code semantics of the program execution (Ma et al., 2022), while the DCFG further explores a better representation of the changed semantics for the modifications. We propose a new framework termed capturing the context-aware code change for commit message generation (C^4MG) to capture the changed semantics based on DCFG. To our knowledge, this is the first attempt to focus on the trade-off between code differences and code context in code change learning. Extensive experiments are conducted and the results suggest the effectiveness of our approach. A new benchmark is constructed to expand the application scope of the approach to file-level code changes. To summarize, we make the following major contributions:

- We argue that it is crucial in commit message generation to reserve the dependency between code difference and the code context while avoiding being misled by the overwhelming amount of unchanged code context.
- We design a novel dynamic control flow graph to extract the changed semantics by emphasizing the local code difference at the block level while leveraging the code context to reserve the dependent relationship at the abstract level.
- We propose a new commit message generation technique termed C^4MG, which captures the changed semantics compactly and adequately based on DCFG. Extensive experiments demonstrate that C^4MG outperforms not only the state-of-the-art open-

source models but also the large language models (e.g., LLaMA3, GPT-4o, and Gemini) on the automatic commit message generation task.

2 Related works

The previous works on commit message generation can be categorized as rule-based, retrieval-based, and learning-based techniques.

The rule-based techniques process code commits with pre-defined patterns or templates to model the connections between code changes and natural languages (Buse and Weimer, 2010; Shen et al., 2016; Vásquez et al., 2015; Moreno et al., 2013). For example, Cortés-Coy et al. (2014) processed the code commit as commit stereotype and type of changes with metrics to fill the pre-defined template of commit messages.

The retrieval-based approaches utilize informational retrieval techniques to adopt existing commit messages with similar code commits (Huang et al., 2017; Liu et al., 2018; Hoang et al., 2020). However, the manually specified rules or templates in rule-based techniques can not work effectively for the code changes that may not apply to the rules. The retrieval-based techniques are limited by the retrieved database, which can only provide existing commit messages as output, rather than generating new ones.

Recently, some researchers have tried to generate commit messages through learning-based techniques (Liu et al., 2019, 2020; Jung, 2021; Nie et al., 2021; Wang et al., 2024; Tao et al., 2024). For example, Jiang et al. (2017) and Loyola et al. (2017) applied machine translation models to encode the code difference and decoded commit messages using LSTM. Inspired by the remarkable achievements in learning programs from the structural perspective (Sun et al., 2019; Bui et al., 2021; Xie et al., 2021), several researchers captured the code change with the structural difference. Dong et al. (2022) constructed a novel code change graph based on the ASTs, along with the edit operation relationships from the old version to the new version to learn the semantic features. However, these works ignore the important code context. Although Xu et al. (2022) on the GumTree (Falleri et al., 2014) tool attempted to import code context beyond code difference, they were modeled by independent encoders and then concatenated as inputs of the decoder. It is still limited by setting them as equally important, and the huge code context would overwhelm the tiny code difference in generation.

3 Dynamic control flow graph

To reserve the dependency between the code difference and the code context while avoiding the redundancy of the code context, the *intra-block* dependency of changed code blocks should be amplified. In contrast, the *inter-block* dependency should be considered at the abstract level. Therefore, we design DCFG, which is a global–local graph for compactly representing code changes in one commit. As illustrated in Fig. 1, one code change can be unfolded to a DCFG including an abstract control flow graph as the backbone and block-level control flow graphs as the fine-grained expansion. Benefiting from the control flow paths that reflect the statement-level program behavior, the control flow graph provides a better representation of code semantics of the program execution (Ma et al., 2022), while the DCFG further explores a better representation of the changed semantics for the modifications.

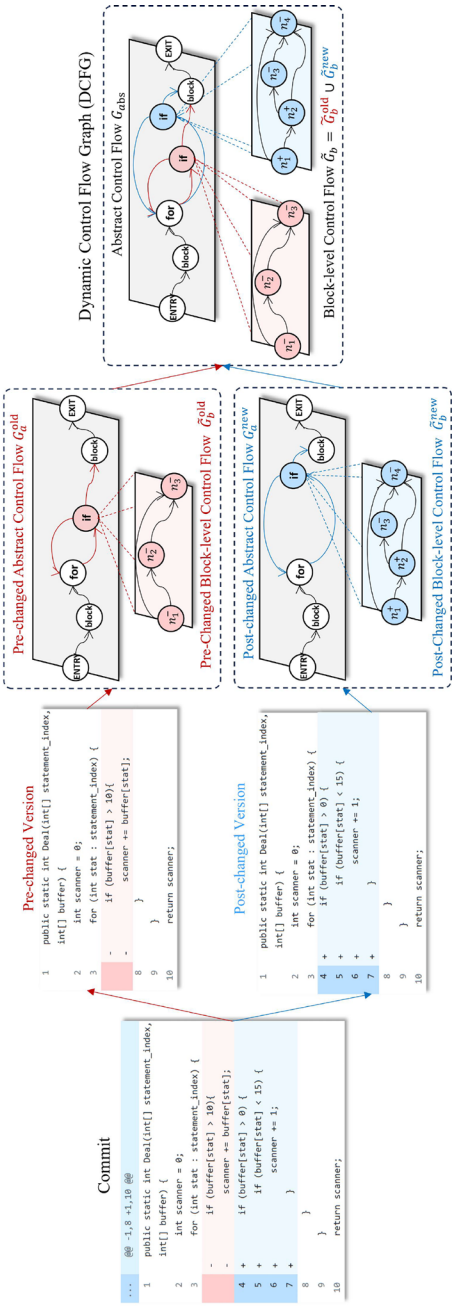

Fig. 1 An example of constructing the DCFG of one commit, where each node of abstracted control flow can be explained to a block-level control flow

Obtaining two versions of pre-changed and post-changed code files from one code commit can be achieved by tools such as *git diff* to perform line-by-line comparisons, resulting in code difference indicated by markers (e.g., '+', '-') denoting added or deleted lines and code context without markers, respectively. For single-file changes, code modifications can be one or multiple disjoint parts. Modeling all code in the file is theoretically reasonable, but in reality, code files can be large, containing thousands of lines, and not all context is relevant to code changes, resulting in noise. To adapt to different sizes of code files, we employ program analysis techniques to extract relevant context. This method utilizes variable analysis and call graph analysis. Variable analysis extracts all variable names from the code differences and matches them within the file. Code blocks containing any variable from the set are preserved and extracted. The call graph obtained by statistic analysis represents the relationship between methods using a directed graph, where nodes represent methods, and the directed edge (a, b) indicates that method a calls method b. cIn the case of Java, specific bytecode instructions may be triggered by some method invocations, such as *invokestatic* for static method calls and *invokeinterface* for interface method calls. By locating these instructions, the positions of the called methods are determined. All successor nodes of modified methods in the file (i.e., all nodes called by the modified method) are retained as relevant context. The extracted parts are concatenated in the original order as subsequent input.

Both pre-changed and post-changed versions of code files are transformed into corresponding old control flow graph: G_0^{old} and new control flow graph G_0^{new}. For nodes in G_0^{old}, if the current node belongs to a compound statement (e.g., for loops, if statements, switch statements) without including other compound statements internally, nodes within the compound statement are merged and abstracted into a new node in the pre-changed abstract CFG G_a^{old}: (N_a^{old}, E_a^{old}), inheriting the connectivity with internal and external nodes, and the dependency among the merged nodes are stored as one instance of the old block-level CFG set $\widetilde{G}_b^{old}$. A similar abstract logic is applied to G_0^{new} to obtain the post-changed abstract CFG G_a^{new}: (N_a^{new}, E_a^{new}), where the new block-level CFG set is $\widetilde{G}_b^{new}$. Then the set of all the block-level CFGs is $\widetilde{G}_b = \widetilde{G}_b^{old} \bigcup \widetilde{G}_b^{new} = \{G_b^1, G_b^2, \cdots G_b^l\}$, where G_b^i is the i-th block-level CFG. The resulting abstract control flow combine the graph structure of G_a^{old} and G_a^{new}, represented as G_{abs}:(N_{abs}, E_{abs}).

4 The proposed approach (C^{4}MG)

As most prior methods mainly focus on the code difference of single granularity, they are hard to directly adopt to capture the changed semantics guided by the global–local structure DCFG. Thus, a new approach C^{4}MG is proposed to capture the semantics of the code change with code context at the abstract level, while comparing the code difference at a fine-grained level. As illustrated in Fig. 2, for one code change, the changed code blocks and the related code context are extracted and further parsed into DCFG, then the dynamic encoder is designed to model the context-aware code change, guided by the dependency on DCFG. Moreover, a context invariance constraint is designed to maintain the context semantic consistency, making the training reasonable and controllable.

The output of the dynamic encoder module serves as the input to the Transformer decoder along with a classification fully connected layer, which also incorporates a pointer network to obtain the probability distribution for the final sequence prediction. The decoder makes auto-regressive predictions, and the predictions are compared to the ground truth using

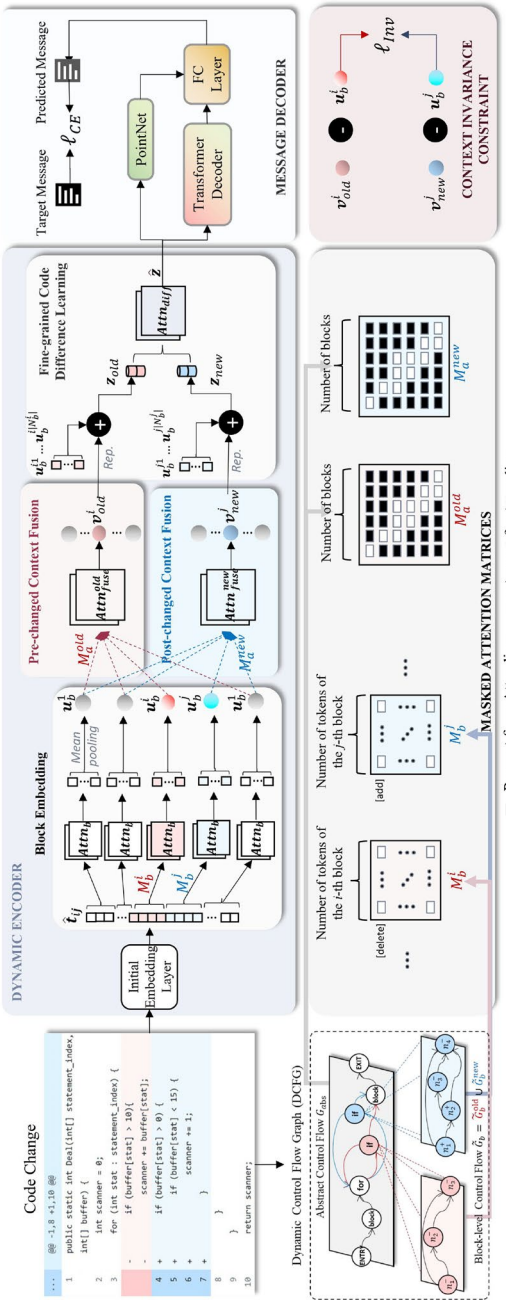

Fig. 2 The overall framework of the C^{4}MG, which takes the commit as input, utilizes dynamic encoder and generates commit message by message decoder. The dynamic encoder is guided by masked attention matrics obtained from DCFG and constrained by context invariance

cross-entropy loss. The commit message generation loss $\mathcal{L}_{CE}$, along with the context invariance loss $\mathcal{L}_{Inv}$ of the context invariance constraint, is combined for joint training:

$$\mathcal{L} = \mathcal{L}_{CE} + \alpha \mathcal{L}_{Inv}, \tag{1}$$

while α is the trade-off hyper-parameter.

4.1 Dynamic encoder

At first, each token of the commit is embedded by the initial embedding layer as follows:

$$\mathbf{t}_{ij}^{k} = Embedding(t_{ij}^{k}), \forall i \in [1, |N_{abs}|], j \in [1, |N_b^i|], k \in [1, K_j^i], \tag{2}$$

where $|N_{abs}|$ is the number of blocks of the commit, $|N_b^i|$ is the number of nodes in the i-th block-level CFG, K_j^i is the number of tokens in the j-th node of the i-th block. In practice, an additional special token in {[*add*], [*delete*], [*context*]} is added at the beginning of each statement, which is similar to the [*cls*] token used in BERT (Devlin et al., 2019).

To capture the changed semantics based on DCFG, three kinds of dependency are utilized to guide the dynamic encoder with structural-guided attention:

1) *intra-block* dependency, which maintains the interaction between the nodes in one block based on the block-level CFG;
2) *pre-changed inter-block* dependency, which maintains the interaction between the pre-changed blocks and code context based on the pre-changed abstract CFG;
3) *post-changed inter-block dependency*, which maintains the interaction between the post-changed blocks and code context based on the post-changed abstract CFG.

4.1.1 Block embedding

The block embedding captures the *intra-block* dependency of the block-level CFG. The statements of code files of the commit are initially encoded as statements embedding. For the nodes of the block-level CFG, the statements of the same node are spliced together as one sequence. A self-attention network $Attn_b$ is leveraged to model each block-level CFG. The j-th node of N_b^i can unfold into a sequence $\hat{\mathbf{t}}_{ij} = [\mathbf{t}_{ij}^1, \mathbf{t}_{ij}^2, \cdots, \mathbf{t}_{ij}^{K_j^i}]$ following the order in the source code. Then the i-th block of N_{abs} can be unfolded into a sequence by concatenating the embeddings of nodes: $[\hat{\mathbf{t}}_{i1}, \hat{\mathbf{t}}_{i2}, \cdots, \hat{\mathbf{t}}_{i|N_b^i|}]$. For each block, a self-attention module is applied to the sequence of the nodes to learn the *intra-block* dependency of corresponding block-level CFGs. The representation of the i-th block is computed as follows:

$$\mathbf{u}_b^{i1}, \mathbf{u}_b^{i2}, \cdots, \mathbf{u}_b^{i|N_b^i|} t= Attn_b([\hat{\mathbf{t}}_{i1}, \hat{\mathbf{t}}_{i2}, \cdots, \hat{\mathbf{t}}_{i|N_b^i|}], M_b^i), \forall i \in [1, |N_{abs}|]. \tag{3}$$

To leverage the *intra-block dependency* of the block-level CFG in G_b^i, we add a masking matrix M_b^i on the attention weight of the corresponding $Attn_b$. The value of the masking matrix M_b^i relies on the node dependency in the block-level CFG. As shown in Fig. 2, each row in the masking matrix denotes the attention masking of other nodes to the corresponding node, where only nodes with white boxes placed allow being attended to. Formally, the intra-block masked attention matrix of the i-th block-level CFG is formulated as M_b^i:

$$M_b^i[p,\ q] = \begin{cases} 0 & p,\ q \in \{[add], [delete], [context]\}\ or \\ & p,\ q \text{ belong to the same node } or \\ & e_{p,q} \in E_b^i; \\ -\infty & otherwise, \end{cases} \tag{4}$$

where p, q are any two tokens in the block, $e_{p,q}$ is the edge between the node which p belongs to and the node which q belongs to, the set E_b^i indicates the relation between the nodes in the i-th block-level CFG G_b^i, and $e_{p,q} \in E_b^i$ means there is a direct edge from the node which p belongs to to the node which q belongs to. Specifically, the masked attention function blocks the transmission of unrelated tokens by setting the attention score to infinitely negative.

Subsequently, the mean pooling is applied to unify the features of all nodes of one block as the representation of the block in the following context fusion:

$$\mathbf{u}_b^i = \frac{1}{|N_b^i|} \sum_{l \in [1, |N_b^i|]} \mathbf{u}_b^{il},\ \forall i \in [1, |N_{abs}|], \tag{5}$$

where $\mathbf{u}_b^i$ denotes the representation of the i-th block of the abstract control flow of DCFG.

4.1.2 Pre-changed context fusion

As the code difference has a dependency on code context outside changed blocks, the semantics of the pre-changed blocks are captured more adequately with interaction with the code context. The pre-changed context fusion captures the *pre-changed inter-block* dependency of the abstract CFG. As illustrated in Fig. 2, the representations of blocks belonging to pre-changed code files are filtered and combined as the sequence $[\mathbf{u}_b^1, \mathbf{u}_b^2, \cdots, \mathbf{u}_b^{|N_{abs}|}]$, which involves $|N_a^{old}|$ nodes of G_a^{old}. A self-attention network $Attn_{fuse}^{old}$ is leveraged to fuse the semantics based on the pre-changed abstract-level control flow. The pre-changed abstract embedding can be defined as:

$$\left[\mathbf{v}_{old}^1, \mathbf{v}_{old}^2, \cdots, \mathbf{v}_{old}^{|N_{abs}|}\right] = Attn_{fuse}^{old}\left([\mathbf{u}_b^1, \mathbf{u}_b^2, \cdots, \mathbf{u}_b^{|N_a^{abs}|}], M_a^{old}\right). \tag{6}$$

Suppose one block belongs to N_a^{old} but not belong to N_a^{new}, the context-ware embedding of this block can be obtained by interaction with other blocks by the pre-changed context fusion. We add a masking matrix M_a^{old} on the attention weight of $Attn_{fuse}^{old}$. The value of the masking matrix M_a^{old} relies on the *pre-changed inter-block dependency* in the pre-changed abstract block CFG G_a^{old}. Formally, the pre-changed inter-block masked attention matrix of the pre-changed abstract CFG G_a^{old} is formulated as M_a^{old}:

$$M_a^{old}[p,\ q] = \begin{cases} 0 & e_{p,q} \in E_a^{old}; \\ -\infty & otherwise, \end{cases} \tag{7}$$

where p, q are any two blocks in the pre-changed abstract CFG, $e_{p,q}$ is the edge between the block which p belongs to and the block which q belongs to, the set E_a^{old} indicates the relation between the blocks in the pre-changed abstract CFG G_a^{old}, and $e_{p,q} \in E_a^{old}$ means there is a direct edge from the block p to the block q.

4.1.3 Post-changed context fusion

The post-changed context fusion captures the *post-changed inter-block* dependency of the abstract CFG. Similarly, the representations of blocks belonging to post-changed code files are filtered and combined as the node sequence $[\mathbf{u}_b^1, \mathbf{u}_b^2, \cdots, \mathbf{u}_b^{|N_{abs}|}]$, which involves $|N_a^{new}|$ nodes of G_a^{new}. A self-attention network $Attn_{fuse}^{new}$ is leveraged to fuse the semantics based on the post-changed abstract-level control flow. The post-changed abstract embedding can be defined as:

$$\left[\mathbf{v}_{new}^1, \mathbf{v}_{new}^2, \cdots, \mathbf{v}_{new}^{|N_{abs}|}\right] = Attn_{fuse}^{new}\left([\mathbf{u}_b^1, \mathbf{u}_b^2, \cdots, \mathbf{u}_b^{|N_{abs}|}], M_a^{new}\right). \tag{8}$$

Similarly, suppose one block belongs to N_a^{new} but does not belong to N_a^{old}, the context-ware embedding of this block can be obtained by interaction with other blocks by the post-changed context fusion. We add a masking matrix M_a^{new} on the attention weight of $Attn_{fuse}^{new}$. The value of the masking matrix M_a^{new} relies on the *post-changed inter-block dependency* in the post-changed abstract block CFG G_a^{new}. Formally, the post-changed inter-block masked attention matrix of the post-changed abstract CFG G_a^{new} is formulated as M_a^{new}:

$$M_a^{new}[p,\ q] = \begin{cases} 0 & e_{p,q} \in E_a^{new}; \\ -\infty & otherwise, \end{cases} \tag{9}$$

where p, q are any two blocks in the post-changed abstract CFG, $e_{p,q}$ is the edge between the block which p belongs to and the block which q belongs to, the set E_a^{new} indicates the relation between the blocks in the post-changed abstract CFG G_a^{new}, and $e_{p,q} \in E_a^{new}$ means there is a direct edge from the block p to the block q.

4.1.4 Fine-grained code difference learning

Fine-grained code difference learning is designed to amplify the code change by comparing the fine-grained representations of the code difference while superimposing the context-aware representations of the changed blocks. Usually, there is more than one changed block, all the changed blocks are concatenated in the order of them in the source code and superposed by the corresponding context-aware vectors.

For instance, as illustrated in Fig. 2, the i-th block is changed to the j-th block of the DCFG. The pre-changed and post-changed context-aware vectors $\mathbf{v}_{old}^i$ and $\mathbf{v}_{new}^j$ are replicated and superposed on the corresponding representations of the pre-changed block and post-changed block. We define $[\mathbf{z}_{old}^1, \mathbf{z}_{old}^2, \cdots, \mathbf{z}_{old}^{|N_b^i|}]$ and $[\mathbf{z}_{new}^1, \mathbf{z}_{new}^2, \cdots, \mathbf{z}_{new}^{|N_b^j|}]$ as the fine-grained representations of the pre-changed block and post-change block.

$$\mathbf{z}_{old}^l = \mathbf{v}_{old}^i + \mathbf{u}_b^{il}, \forall l \in [1, |N_b^i|], \tag{10}$$

$$\mathbf{z}_{new}^l = \mathbf{v}_{new}^j + \mathbf{u}_b^{jl}, \forall l \in [1, |N_b^j|]. \tag{11}$$

Then a self-attention network $Attn_{diff}$ is leveraged to capture the code difference.

$$\hat{\mathbf{z}} = Attn_{diff}\left([\mathbf{z}_{old}^1, \mathbf{z}_{old}^2, \cdots, \mathbf{z}_{old}^{|N_b^i|}, \mathbf{z}_{new}^1, \mathbf{z}_{new}^2, \cdots, \mathbf{z}_{new}^{|N_b^j|}]\right). \tag{12}$$

4.2 Message decoder

The latent representations obtained by the dynamic encoder module in Sect. 4.1 are denoted as $\hat{\mathbf{z}}$, which serve as inputs to the message decoder module. The C^{4}MG adopts a standard Transformer decoder consisting of multiple decoder layers and a classification fully connected layer. The probability distribution predicted by the decoder is combined with the copy probability distribution obtained from the pointer network to obtain the final probability distribution for prediction. Consistent with the baselines, which used cross-entropy loss to compute the loss function for the sequence of tokens, the commit message generation loss is as follows:

$$\begin{aligned} \mathcal{L}_{CE} &= -\log P(\mathbf{y}|\hat{\mathbf{z}};\theta) \\ &= -\sum_{t=1}^{m} \log P(\mathbf{y}_t|\hat{\mathbf{y}}_{<t}, \hat{\mathbf{z}};\theta), \end{aligned} \tag{13}$$

where m is the length of the message, $\mathbf{y}_t$ is the t-th token embedding of the ground truth, and $\hat{\mathbf{y}}_{<t}$ is the first $t-1$ predicted token embeddings of the generated message.

4.3 Context invariance constraint

Abstract control flow and block-level control flow not only have a global–local relation but also satisfy an additional constraint, that is the semantics imported by code context to the pre-changed block and post-changed block should be consistent in the pre-changed context fusion and post-changed context fusion. As defined above, the i-th pre-changed block is represented as $\mathbf{u}_b^i$, and the j-th post-changed block is represented as $\mathbf{u}_b^j$. $\mathbf{v}_{old}^i$ denotes the representation of the i-th pre-changed block after pre-changed context fusion, and $\mathbf{v}_{new}^j$ denotes the representation of the j-th post-changed block after post-changed context fusion. Then the following constraint holds:

$$\mathbf{v}_{old}^i - \mathbf{u}_b^i = \mathbf{v}_{new}^j - \mathbf{u}_b^j. \tag{14}$$

To introduce this constraint into the model, this section introduces a context invariance loss $\mathcal{L}_{Inv}$ that leverages the module to bridge the representations of both sides of the equation in a high-dimensional feature space, which is calculated based on the mean square loss:

$$\mathcal{L}_{Inv} = \left(\left(\mathbf{v}_{old}^i - \mathbf{u}_b^i \right) - \left(\mathbf{v}_{new}^j - \mathbf{u}_b^j \right) \right)^2. \tag{15}$$

In the high-dimensional feature space, the context invariance constraint can reduce the vector distance from all directions to preserve the constraint of consistent code context, making the training reasonable and controllable.

5 Experiment

To evaluate our proposed model, we conduct extensive experiments on the large dataset. In this section, we elaborate on the datasets, baselines, metrics and implementation, and experiment results.

5.1 Dataset

Although a few datasets of commit message generation have been proposed, a limitation of most datasets is only contain code differences and lack of context. Among the few datasets that include context (Xu et al., 2022), the granularity of code changes is very small, typically limited to individual code blocks or primarily focused on changes within a single function. However, code modifications exhibit hierarchical relationships, where each code change can involve multiple file modifications, and each file may have multiple block-level modifications, further extending to line-level and token-level changes. To model the file-level code changes, a completely new dataset is constructed to accommodate the more general application scenarios.

The file-level dataset is collected from the open-source version control platform GitHub. After the filtering and preprocessing as described in the Appendix, a total of 110,116 commit-message pairs are reserved in the dataset. To facilitate training deep learning models, all data in the dataset are randomly shuffled, with 80% as the train set, 10% as the validation set, and 10% as the test set. The statistics of the proposed dataset are shown in Table 1.

5.2 Baselines

The start-of-the-art methods in commit message generation are compared.

- NMT (Jiang et al., 2017; Loyola et al., 2017; Hal, 2019) adopts attention-based RNN encoder-decoder models to generate commit messages.
- NNGen (Liu et al., 2018) is an information retrieval approach. It re-uses the message from the most similar code change for a given code change.
- CoDiSum (Xu et al., 2019) models both code skeleton and code semantics, and it also includes the copy mechanism.
- ATOM (Liu et al., 2020) models syntactic regularities by encoding the paths between leaf nodes in the ASTs, but does not consider the naturalness of code.
- CoMEG (Xu et al., 2022) models code context and code difference by independent encoders then concatenated as equal important inputs of the decoder.
- COMU (Wang et al., 2024) models the code change by extracting multi-grained information from the changed code at the line and AST levels.
- KADEL (Tao et al., 2024) builds a commit knowledge model and designs a novel dynamic denoising training method to achieve more effective training.

Table 1 The statistics of the proposed dataset, where avg-commit is the average length of the commits, avg-diff is the average length of the changed blocks, avg-block is the average number of the blocks in one commit, avg-node is the average number of the nodes in one block, and avg-msg is the average length of the commit messages

partition	number	avg-commit	avg-diff	avg-block	avg-node	avg-msg
training	88090	271.05	13.37	24.00	1.81	8.36
validation	11013	270.57	13.70	23.93	1.79	8.37
testing	11013	268.23	13.12	23.91	1.79	8.39

In addition, several widely used large pre-trained models in software mining are compared as a straightforward way to treat code context and code difference as input equally, including CodeBERT (Feng et al., 2020), DOBF (Lachaux et al., 2021), and GraphCodeBERT (Guo et al., 2021), which are encoder-only model and they are spliced onto the same decoder as our method to be fine-tuned. And the encoder-decoder model CodeT5 (Wang et al., 2021) is also fine-tuned as one baseline. All the tokens including code differences and code contexts are inputted into the pre-trained model separated by the sign tokens. Moreover, the advanced LLMs including DeepSeek-Coder (Guo et al., 2024), CodeQwen (Bai et al., 2023), LLaMA3 (Touvron et al., 2023), GPT−3.5 (Ouyang et al., 2022), GPT-4o (Achiam et al., 2023), and Gemini (Anil et al. (2023) are also evaluated on the dataset. For the LLMs, the proper prompts are given to elicit desired responses for the commit message generation.

5.3 Metrics and implementation

We employ three widely used metrics in the field of natural language generation to evaluate the quality of the generated results. BLEU is the most popular lexical similar metric in natural language processing, originated in 2002 from the machine translation research community (Papineni et al., 2002). BLEU works by comparing n-grams in the prediction and reference, with a penalty for overly short sentences. BLEU-1/2/3/4 correspond to the scores of unigram, 2-grams, 3-grams, and 4-grams, respectively.

$$\text{BLEU-N} = BP \times \exp\left(\sum_{n=1}^{N} w_n \log p_n\right), \text{N} \in \{1, 2, 3, 4\} \tag{16}$$

where the weights $w_n = \frac{1}{N}$, c, r are the lengths of the candidate and reference sequences respectively, BP equals $\exp(1 - r/c)$ if $c \leq r$, otherwise it is 1. Rouge-L provides an F-score based on the longest common sub-sequence (LCS), which compares the similarity between two given texts. Meteor indicates a weighted F-score based on mapping unigrams and a penalty function for incorrect word order, where F_{mean} is computed with the unigram precision (P) and the unigram recall (R):

$$\text{Meteor} = F_{mean} \times (1 - \text{Penalty}), \tag{17}$$

$$F_{mean} = \frac{10\text{PR}}{\text{R} + 9\text{P}}. \tag{18}$$

Penalty is levied for fragmented matches as the ratio of matched chunk number to match the unigram number.

In the experimental setup, the embedding dimension for the vocabulary is set to 512, and α is set as 0.1. During the training phase, the maximum number of epochs is set to 50, and early stopping is employed to prevent overfitting. The initial embedding layer is initialized by the parameters of CodeBERT (Feng et al., 2020) to accelerate the training process. The $Attn_b$, $Attn^{\cdot}_{\text{fuse}}$ and $Attn^{\cdot}_{\text{diff}}$ are 2 layer attention networks, and each layer contains 8 attention heads, with a hidden embedding dimension of 512 and a feed-forward dimension of 2048. The decoders consist of 6 stacked layers, with each layer containing 8 attention heads. During the training phase, the maximum number of epochs is set to 50, and early stopping is employed to prevent overfitting. Loshchilov and Hutter (2017) optimizer and a multi-step learning rate scheduler are used, with an initial learning rate of $2e{-}4$. The

training batch size is 64 with a dropout rate of 0.1. All models are built using PyTorch on the Ubuntu operating system and trained on 8 Nvidia Tesla V100 GPUs. All the baselines are trained on the training set, and the hyper-parameters are chosen with the best performance by the validation set.

5.4 Overall performance comparison

Table 2 presents the performance of baselines on the dataset. The best experimental result for each metric is highlighted in bold. Overall, C^4MG achieves significant improvements on all metrics and outperforms the state-of-the-art approach. Moreover, although AST-based approaches such as ATOM, CoMEG, and COMU demonstrated notable improvements compared to sequence models, the control-flow-based model in this paper generally exhibited superior performance. This indicates the dynamic control flow graph is more effective than existing code structures for modeling the dynamic code changes in commit message generation, and validates the importance of program execution information in code understanding, which provides significant insights for future work.

Table 3 presents the performance of pre-trained models and large language models on the dataset. The pre-train models are fine-tuned on the same training dataset as our method, and the large language models are guided by the proper prompt. With the number of parameters increasing, the performance of large language models in commit message generation is gradually growing. But compared with the two best performing large language models, GPT-4o and Gemini, the C^4MG have achieved comparable performance with fewer parameters in the specific task. The Student's t-test is conducted between our technique and other baselines in Table 2 and Table 3, and the results show that the improvements are significant with $p < 0.01$.

5.5 Ablation study

To validate the effectiveness of different components, various configurations of the model are created by removing specific components. Table 4 presents the performance of different configurations of the C^4MG. Each of the components in the dynamic encoder is removed individually to observe the changes in model performance. The

Table 2 The experimental results of the start-of-the-art methods of commit message generation

Method	BLEU-1	BLEU-2	BLEU-3	BLEU-4	Rouge-L	Meteor
NMT (Loyola et al., 2017)	16.170	14.430	12.974	13.524	12.320	4.434
NNGen (Liu et al., 2018)	18.735	17.179	17.256	17.949	12.500	6.551
CoDiSum(Xu et al., 2019)	11.796	10.457	8.256	8.008	7.020	2.725
CoMEG (Xu et al., 2022)	15.717	14.040	12.777	13.440	11.385	4.453
ATOM (Liu et al., 2020)	19.168	17.637	16.517	17.393	12.215	6.568
COMU (Wang et al., 2024)	16.701	14.745	11.536	10.995	7.870	2.410
KADEL (Tao et al., 2024)	19.679	18.783	17.020	16.772	12.389	6.667
C^4MG	22.486	21.182	20.387	21.497	15.021	8.731
Improvement (w.r.t best)	+14.3%	+12.8%	+18.1%	+19.8%	+20.2%	+31.0%

Table 3 The experimental results of the pre-trained models and large language models

Method	BLEU-1	BLEU-2	BLEU-3	BLEU-4	Rouge-L	Meteor
Pre-trained Models						
DOBF	13.491	12.048	10.117	10.126	9.657	3.189
CodeBERT	16.978	15.335	13.979	14.604	13.472	4.951
GraphCodeBERT	18.486	17.006	17.820	16.136	13.396	6.473
CodeT5	19.527	17.820	18.486	17.006	13.406	5.333
Large Language Models						
DeepSeek-Coder-1.3B	6.285	5.193	5.016	5.073	3.591	2.412
DeepSeek-Coder-6.7B	17.069	13.885	13.233	13.454	9.380	5.837
CodeQwen1.5-7B	12.888	10.374	9.880	9.946	6.587	2.285
LLaMA3-8B	15.123	11.554	10.585	10.427	10.041	6.458
LLaMA3-70B-Instruct	18.049	14.637	13.700	13.524	10.365	7.173
GPT−3.5	19.637	16.196	15.657	16.225	11.028	6.100
GPT-4o	20.150	16.613	15.488	15.749	14.078	7.912
Gemini	21.555	17.712	17.775	19.677	12.083	7.030
C^4MG	22.486	21.182	20.387	21.497	15.021	8.731
Improvement (w.r.t best)	+4.3%	+18.9%	+10.3%	+9.2%	+6.7%	+10.4%

Table 4 The performance of different configurations of C^4MG in terms of BLEU-1/2/3/4, Rouge-L, and Meteor

Method	BLEU-1	BLEU-2	BLEU-3	BLEU-4	Rouge-L	Meteor
w/o Block Embedding	17.642	15.988	16.624	17.152	12.591	6.240
w/o Pre Context Fusion	18.176	18.213	16.153	16.681	13.467	6.267
w/o Post Context Fusion	18.697	16.965	17.415	18.838	14.516	6.860
w/o Fine-grained	17.091	15.135	15.481	16.785	13.457	6.179
w/o Context Invariance	19.245	17.837	17.297	18.983	13.407	7.035
C^4MG	22.486	21.182	20.387	21.497	15.021	8.731

removal of any of the components indeed caused a performance decrease, indicating the rationality and effectiveness of these components. Among them, the experimental results show that removing the fine-grained code difference learning resulted in the largest decline. It is because, in the configuration of *w/o* Fine-grained., all the embeddings of pre-changed context fusion and post-changed context fusion are concatenated as the input of the message decoder without superposed on the changed blocks representations, which causes the model to mislead by the overwhelming amount of the code context and ignore the subtle changes. The phenomenon has supported the motivation of our work. Moreover, it can be observed that removing the context invariance leads to a decline in all metrics. This module utilizes constraints to further restrict the invariance of code context in the dynamic modeling process, preventing feature representations from becoming scattered.

6 Human evaluation

To further study the quality of generated commit messages from the perspective of developers, we perform a human evaluation. We compare C^{4}MG with the retrieval-based technique NNGen and the learning-based technique GPT−3.5. We invite ten developers to participate in this study, who have experience in Java programming language ranging from 1 to 3 years.[3]

6.1 Study design

Following previous works (Liu et al., 2020; Dong et al., 2022), we randomly select 100 commits from the testing set and design a questionnaire for human evaluation. For each commit, the questionnaire includes the code change, the ground truth commit message, and the commit messages generated by C^{4}MG as well as the compared techniques. Each invited participant is asked to score the commit messages generated by three techniques. The score ranges from 0 to 4, and a higher score indicates a higher similarity between the generated commit message and the ground truth commit message. We follow the existing scoring criterion (Dong et al., 2022), and a detailed definition is reported in Table 5. To avoid bias, all three techniques are anonymous in the questionnaire and each participant fills in the questionnaire separately.

6.2 Human evaluation results

For each technique, we measure the quality of its generated commit message based on the average scores of ten participants. As shown in Table 5, C^{4}MG exhibits the largest ratio of high-quality commit messages while the lowest ratio of low-quality commit messages. The average score indicates the out-performance of C^{4}MG over NNGen and GPT−3.5 techniques. Moreover, the variance of the scores is inferior, which means that the participants agreed on the higher quality of the commit messages generated by C^{4}MG and the lower quality by NNGen. To confirm our observations, we further conducted the Student's t-test between the scores of C^{4}MG and the other techniques. The results further confirm that the difference is significant with $p < 0.01$.

7 Case study

We further present the case in which C^{4}MG achieves higher scores. The case includes the code change, the ground truth, and the commit messages generated by C^{4}MG and the compared techniques (i.e., NNGen and GPT−3.5).

The example in Fig. 3 shows the code changes for fixing tests to verify debug logging. As shown in the Figure, the messages retrieved by NNGen lack the details of the change. The messages generated by GPT−3.5 capture the details of the code change of the about *debug logging*, but it ignores the scope of the code change, which is abstracted from the Python Decorator `Test` in line 124 of the code context. Compared with other approaches,

[3] None of them are co-authors of this paper.

Table 5 Scoring criterion and evaluation result of human evaluation

Score	Definition
0	Neither relevant in semantics nor having shared tokens.
1	Irrelevant in semantics but with some shared tokens.
2	Partially similar in semantics, but contains exclusive information.
3	Highly similar but not identical in semantics.
4	Identical in semantics.

Model	Low(0~1)	medium(2~3)	high(3~4)	Average Score	Variance
NNGen	48.2%	43.4%	8.4%	1.608	1.346
GPT−3.5	34.2%	53.2%	12.6%	1.964	0.918
C^4MG	30.8%	47.4%	21.8%	2.270	1.042

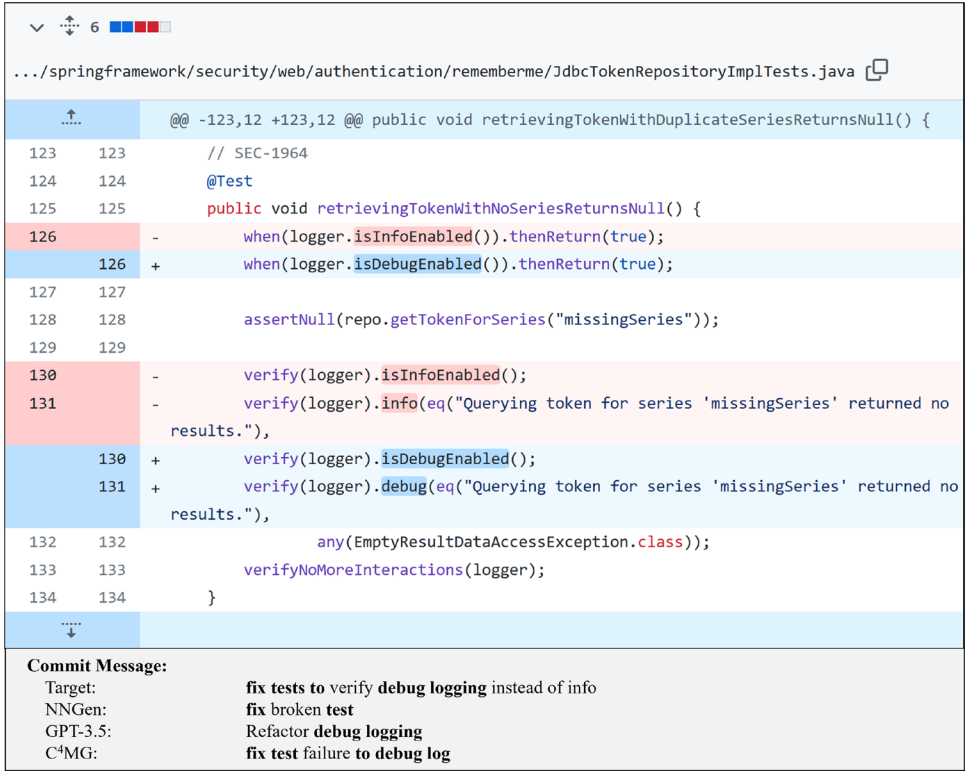

Fig. 3 An example of the commit messages generated by different techniques

C[4]MG not only captures the changed details but also successfully predicts the scope of fixing bugs in `Test`. It has shown the effectiveness of C[4]MG in utilizing the code context to generate commit messages of better quality.

8 Threat analysis

Our results should be interpreted with several threats to validity in mind.

- The internal threat to validity lies in the implementation of compared techniques. To reduce this threat, we directly reuse the implementation of the compared techniques from their reproducible packages and the weights of pre-trained models, if they are available and executable. Otherwise, we reimplement the techniques strictly following their papers. We build our approach on existing mature libraries, such as tree-sitter.[4]
- The external threat to validity lies in the dataset used in the experiment. To mitigate the external threat, the widely-used open-source projects are chosen based on their GitHub stars. Moreover, filtering and preprocessing are performed to ensure no violation case is applicable. The dataset is publicly available.
- The construct threat lies in the metrics used in the evaluation. To reduce this threat, we adopt several metrics that have been widely used by prior work on commit message generation (Xu et al., 2022, 2019). To evaluate the effectiveness from the perspective of developers, we further perform the human evaluation. We strictly follow the procedure of previous work (Liu et al., 2020; Wang et al., 2021) and invite experienced developers, to reduce the threats to human evaluation.

9 Conclusion

This paper focuses on the task of commit message generation to help developers for effective development. We highlight reserving the dependency between code difference and code context in code change learning while avoiding being misled by the overwhelming amount of the code context for enhancing commit message generation. The Dynamic Control Flow Graph (DCFG) is proposed for modeling context-aware code change. A novel framework called C[4]MG is designed, which captures the changed semantics guided by DCFG, leverages code context while avoiding redundancy, and constrains the consistency of code context in the modification. Experimental results demonstrate the effectiveness of our method in improving the quality of generated commit messages. For future work, the trade-off between code differences and code context in code change learning can be further explored and contribute to enhancing the understanding of software evolution.

[4] https://github.com/tree-sitter/tree-sitter.

Appendix

In this section, we elaborate on the preprocessing of the dataset, including filtering Based on commit type, filtering based on verb/direct-object pattern, data denoising, and word splitting.

Filtering based on commit type

Code commits that merge or roll back versions without meaningful changes, unconventional commits like project initialization or basic functionality updates, and consecutive commits with duplicate log messages are filtered out. Non-ASCII character commit messages are also excluded. The first meaningful sentence of each commit message is extracted to represent the code changes concisely.

Filtering based on verb/direct-object pattern

Commit messages describe code changes or intentions and can vary in style and content. This variability can affect the training of deep learning models. Jiang et al. (2017) found that 47% of commit messages follow a verb/direct-object pattern, such as “correct a typo” or “add useful tests.” To identify these patterns, Stanford CoreNLP (Manning et al., 2014) is used to annotate syntactic dependency relations in commit messages. Specifically, dependencies like “obj” and “dobj” are targeted, as seen in the dependency (obj, fix, bug) in “fix the bug.” During data processing, only commit messages starting with “obj” or “dobj” are retained to standardize the writing conventions and minimize formatting impacts on performance.

Data denoising

In complex multi-developer environments, commit messages often include mentions of contributors, reviewers, and bug reporters, introducing noise into the data. Keywords like "reviewed by" and "reported by" are used to filter out this noise. Commit IDs, issue IDs, web links, and colloquial phrases like "thank you" or "oops my bad" are also removed using regular expressions to clean the data.

Spliting into subwords

The names of variables, functions, etc., are often composed of multiple subwords derived from natural language, which should be split into sets of subtokens, e.g., the word “ClassName” is divided into “Class” and “Name”. Out-of-vocabulary words in the code are represented as “UNK”.

Author contributions Yali Du contributed to the writing of this paper and performed the main experiments. Ying Li performed part of the experiments, contributed to the revision of this manuscript, and provided

valuable suggestions for the overall method. Yi-Fan Ma and Ming Li provided valuable feedback, suggestions, and critical revision of the manuscript.

Funding This research was supported by NSFC (62076121, 61921006), Major Program (JD) of Hubei Province (2023BAA024), and Postgraduate Research & Practice Innovation Program of Jiangsu Province (KYCX24_0301).

Data availability https://anonymous.4open.science/r/C4MG.

Declarations

Conflict of interest Not applicable.

Ethics approval Not applicable.

Consent to participate and publication Not applicable.

References

Dong, J., Lou, Y., Zhu, Q., Sun, Z., Li, Z., Zhang, W. & Hao, D. Fira: fine-grained graph-based code change representation for automated commit message generation. In: Proceedings of the 44th international conference on software engineering, pp. 970–981 (2022).

Dyer, R., Nguyen, H.A., Rajan, H. & Nguyen, T.N. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In: Proceedings of the 35th international conference on software engineering, pp. 422–431 (2013).

He, Y., Wang, L., Wang, K., Zhang, Y., Zhang, H. & Li, Z. COME: commit message generation with modification embedding. In: Proceedings of the 32nd international symposium on software testing and analysis, pp. 792–803 (2023).

Wang, L., Tang, X., He, Y., Ren, C., Shi, S., Yan, C. & Li, Z. Delving into commit-issue correlation to enhance commit message generation models. In: Proceedings of the 38th IEEE/ACM international conference on automated software engineering, pp. 710–722 (2023).

Xu, S., Yao, Y., Xu, F., Gu, T., Tong, H. & Lu, J. Commit message generation for source code changes. In: Proceedings of the 28th international joint conference on artificial intelligence, pp. 3975–3981 (2019).

Liu, S., Gao, C., Chen, S., Nie, L. Y., & Liu, Y. (2022). ATOM: Commit message generation based on abstract syntax tree and hybrid ranking. *IEEE Transactions on Software Engineering, 48*(5), 1800–1817. https://doi.org/10.1109/TSE.2020.3038681

Shi, S.-T., Li, M., Lo, D., Thung, F. & Huo, X. Automatic code review by learning the revision of source code. In: Proceedings of the AAAI conference on artificial intelligence, pp. 4910–4917 (2019).

Xu, S., Yao, Y., Xu, F., Gu, T. & Tong, H. Combining code context and fine-grained code difference for commit message generation. In: Proceedings of the 13th Asia-pacific symposium on internetware, pp. 242–251 (2022).

Ma, Y.-F. & Li, M. Learning from the multi-level abstraction of the control flow graph via alternating propagation for bug localization. In: Proceedings of the 22nd IEEE international conference on data mining, pp. 299–308 (2022).

Buse, R.P. & Weimer, W.R.: Automatically documenting program changes. In: Proceedings of the 25th IEEE/ACM International conference on automated software engineering, pp. 33–42 (2010).

Shen, J., Sun, X., Li, B., Yang, H. & Hu, J. On automatic summarization of what and why information in source code changes. In: Proceedings of the IEEE 40th annual computer software and applications conference, pp. 103–112 (2016).

Vásquez, M.L., Cortes-Coy, L.F., Aponte, J. & Poshyvanyk, D. Changescribe: A tool for automatically generating commit messages. In: Proceedings of the 37th IEEE/ACM international conference on software engineering, pp. 709–712 (2015).

Moreno, L., Aponte, J., Sridhara, G., Marcus, A., Pollock, L.L. & Vijay-Shanker, K. Automatic generation of natural language summaries for java classes. In: Proceedings of the 21st International conference on program comprehension, pp. 23–32 (2013).

Cortés-Coy, L.F., Linares-Vásquez, M., Aponte, J. & Poshyvanyk, D. On automatically generating commit messages via summarization of source code changes. In: Proceedings of the 14th international working conference on source code analysis and manipulation, pp. 275–284 (2014).

Huang, Y., Zheng, Q., Chen, X., Xiong, Y., Liu, Z. & Luo, X. Mining version control system for automatically generating commit comment. In: Proceedings of the ACM/IEEE international symposium on empirical software engineering and measurement, pp. 414–423 (2017).

Liu, Z., Xia, X., Hassan, A.E., Lo, D., Xing, Z. & Wang, X. Neural-machine-translation-based commit message generation: how far are we? In: Proceedings of the 33rd ACM/IEEE international conference on automated software engineering, pp. 373–384 (2018).

Hoang, T., Kang, H.J., Lo, D. & Lawall, J. Cc2vec: Distributed representations of code changes. In: Proceedings of the ACM/IEEE 42nd international conference on software engineering, pp. 518–529 (2020).

Liu, Q., Liu, Z., Zhu, H., Fan, H., Du, B. & Qian, Y. Generating commit messages from diffs using pointer-generator network. In: Proceedings of the IEEE/ACM 16th international conference on mining software repositories, pp. 299–309 (2019).

Jung, T. Commitbert: Commit message generation using pre-trained programming language model. arXiv preprint arXiv:2105.14242 (2021).

Nie, L. Y., Gao, C., Zhong, Z., Lam, W., Liu, Y., & Xu, Z. (2021). CoreGen: Contextualized code representation learning for commit message generation. *Neurocomputing, 459*, 97–107. https://doi.org/10.1016/j.neucom.2021.05.039

Wang, C., Zhang, L., & Zhang, X. (2024). Multi-grained contextual code representation learning for commit message generation. *Information and Software Technology, 167*, 107393.

Tao, W., Zhou, Y., Wang, Y., Zhang, H., Wang, H., & Zhang, W. (2024). Kadel: Knowledge-aware denoising learning for commit message generation. *ACM Transactions on Software Engineering and Methodology., 33*, 1–32.

Jiang, S., Armaly, A. & McMillan, C. Automatically generating commit messages from diffs using neural machine translation. In: Proceedings of the 32nd IEEE/ACM international conference on automated software engineering, pp. 135–146 (2017).

Loyola, P., Marrese-Taylor, E. & Matsuo, Y. A neural architecture for generating natural language descriptions from source code changes. arXiv preprint arXiv:1704.04856 (2017).

Sun, Z., Zhu, Q., Mou, L., Xiong, Y., Li, G. & Zhang, L. A grammar-based structural cnn decoder for code generation. In: Proceedings of the 33rd AAAI conference on artificial intelligence, pp. 7055–7062 (2019).

Bui, N.D.Q., Yu, Y. & Jiang, L. Treecaps: Tree-based capsule networks for source code processing. In: Proceedings of the 35th AAAI conference on artificial intelligence, pp. 30–38 (2021).

Xie, B., Su, J., Ge, Y., Li, X., Cui, J., Yao, J. & Wang, B. Improving tree-structured decoder training for code generation via mutual learning. In: Proceedings of the 35th AAAI conference on artificial intelligence, pp. 14121–14128 (2021).

Falleri, J.-R., Morandat, F., Blanc, X., Martinez, M. & Monperrus, M. Fine-grained and accurate source code differencing. In: Proceedings of the 29th ACM/IEEE International conference on automated software engineering, pp. 313–324 (2014).

Devlin, J., Chang, M., Lee, K. & Toutanova, K. BERT: pre-training of deep bidirectional transformers for language understanding. In: Proceedings of the 2019 Conference of the North American chapter of the association for computational linguistics, pp. 4171–4186 (2019).

Hal, S. Generating commit messages from git diffs. arXiv:1911.11690 (2019).

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D. & Zhou, M. CodeBERT: A pre-trained model for programming and natural languages. In: Findings of the Association for Computational Linguistics: EMNLP, pp. 1536–1547 (2020).

Lachaux, M., Rozière, B., Szafraniec, M., & Lample, G. (2021). DOBF: A deobfuscation pre-training objective for programming languages. *Advances in Neural Information Processing Systems, 34*, 14967–14979.

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., Tufano, M., Deng, S.K., Clement, C.B., Drain, D., Sundaresan, N., Yin, J., Jiang, D. & Zhou, M. GraphCodeBERT: Pre-training code representations with data flow. In: International conference on learning representations (2021).

Wang, Y., Wang, W., Joty, S. & Hoi, S.C. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In: Proceedings of the 2021 Conference on empirical methods in natural language processing, pp. 8696–8708 (2021).

Guo, D., Zhu, Q., Yang, D., Xie, Z., Dong, K., Zhang, W., Chen, G., Bi, X., Wu, Y., Li, Y.K., Luo, F., Xiong, Y. & Liang, W. Deepseek-coder: When the large language model meets programming - the rise of code intelligence. CoRR **abs/2401.14196** (2024).

Bai, J., Bai, S., Chu, Y., Cui, Z., Dang, K., Deng, X., Fan, Y., Ge, W. & Han, Y., et al. Qwen technical report. arXiv preprint arXiv:2309.16609 (2023).

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F. & et al. Llama: Open and efficient foundation language models. arXiv preprint arXiv:2302.13971 (2023).

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C. L., Mishkin, P., Zhang, C., et al. (2022). Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems, 35*, 27730–27744.

Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F.L. & Almeida, D., et al. Gpt-4 technical report. arXiv preprint arXiv:2303.08774 (2023).

Anil, R., Borgeaud, S., Wu, Y., Alayrac, J. & Yu, J., al. Gemini: A family of highly capable multimodal models. CoRR **abs/2312.11805** (2023).

Papineni, K., Roukos, S., Ward, T. & Zhu, W. Bleu: a method for automatic evaluation of machine translation. In: Proceedings of the 40th Annual meeting of the association for computational linguistics, pp. 311–318 (2002).

Loshchilov, I. & Hutter, F. Decoupled weight decay regularization. arXiv preprint arXiv:1711.05101 (2017).

Wang, H., Xia, X., Lo, D., He, Q., Wang, X., & Grundy, J. (2021). Context-aware retrieval-based deep commit message generation. *ACM Transactions on Software Engineering and Methodology, 30*(4), 1–30. https://doi.org/10.1145/3464689

Manning, C.D., Surdeanu, M., Bauer, J., Finkel, J.R., Bethard, S. & McClosky, D. The stanford corenlp natural language processing toolkit. In: Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations, pp. 55–60 (2014).